Enhancing Java Runtime Environment for Smart Cards Against Runtime Attacks

Raja Naeem Akram(✉), Konstantinos Markantonakis, and Keith Mayes

Information Security Group, Smart Card Centre,
Royal Holloway, University of London, Egham, Surrey, UK
{R.N.Akram,K.Markantonakis,Keith.Mayes}@rhul.ac.uk

Abstract. Smart cards are mostly deployed in security-critical environments in order to provide a secure and trusted access to the provisioned services. These services are delivered to a cardholder using the Service Provider's (SPs) applications on his or her smart card(s). These applications are at their most vulnerable state when they are executing. There exist a variety of runtime attacks that can circumvent the security checks implemented either by the respective application or the runtime environment to protect the smart card platform, user and/or application. In this paper, we discuss the Java Runtime Environment and a potential threat model based on runtime attacks. Subsequently, we discussed the counter-measures that can be deployed to provide a secure and reliable execution platform, along with an evaluation of their effectiveness, incurred performance-penalty and latency.

1 Introduction

An application on a smart card relies on the Smart Card Runtime Environment (SCRT) for secure and reliable execution. An SCRT contains a library of Application Programming Interfaces (APIs) that provide a secure and reliable interface between the installed applications and on-card services. An SCRT is used in order to:

1. Provide a secure and reliable program execution.
2. Enforce an execution isolation and access to memory locations.
3. Provide an interface to access cryptographic algorithms.
4. Protect the platform and applications from malicious or ill-formed applications.
5. Handle communication between applications and with external entities.

In early 2000, fault attacks became the modus operandi of adversaries to subvert the implemented cryptographic algorithms in the smart card industry. Since then the technology has evolved to counter these threats to some extent [3–5]. Althought, the full extent is not publically know, there has been a growing interest in fault injection and combined attacks [6–8] to subvert the protection mechanisms on a smart card. In combined attacks both the software (i.e. attacker's application) and fault injection are used to achieve the objectives. In this paper,

G. Pernul et al. (Eds.): ESORICS 2015, Part II, LNCS 9327, pp. 541–560, 2015.
DOI: 10.1007/978-3-319-24177-7_27

we analyse the attacks that target the SCRT and provide counter-measures. The attacks we have considered in this paper are fault and combined attacks targetted at the SCRT. In this paper, we focus on Java Cards; therefore, we will constantly refer to the Java Card Runtime Environment (JCRE) and it is used synonymously with SCRT. The rationale is that the JCRE has an open specification as compared to alternatives such as Multos, and new attacks mostly target Java Cards.

1.1 Contributions of the Paper

In this paper, we propose and evaluate the following:

1. A JCRE protection framework referred to as the "Runtime Protection Mechanism (RPM)".
2. Inclusion of the application developer's security requirements at the compilation of the application. If these requirements do not violate the security requirements of the JCRE, the runtime environment will try to enforce them.
3. A set of countermeasures that include:
   (a) Operand Stack Integrity: Safeguarding the JCRE's operand stack from any malicious modifications.
   (b) Permitted Execution Path Analysis: Evaluate the program flow and verify whether a particular execution path is allowed or not, based on the security requirements; defined by the application developer and/or JCRE.
   (c) Bytecode Integrity: To verify and validate whether the execution code of an application, in storage (persistent memory) and while in non-persistent memory during execution has not been modified.
4. Two variants of "Runtime Security Manager (RSM)" referred to as serial and parallel mode. The RSM enforces the security requirements defined by the application developers and JCRE as part of the RPM along with deploying the countermeasures. The variants are differentiated based on the architecture of the underlying hardware and the point at which the RSM verify and validate an application during its execution.
5. The proposed framework is implemented, and evaluated for security, incurred performance penalty and latency.

2 Smart Card Runtime Environment

In this section, we open the discussion with a brief description of the Java Card Virtual Machine (JCVM) followed by related work, and our motivation for the paper.

2.1 Java Card Virtual Machine

The JCRE consists of APIs, system classes, Java Card Virtual Machine (JCVM), and native methods. The most crucial component of the JCRE is the JCVM that

actually interpret the application code to execute on the underlying hardware. The architecture of the JCVM is more or less similar between various Java Card versions.

An application is coded in a subset of the Java language that is supported by the JCVM, which is represented as a Java file. The application is then compiled into a class file, and it is packaged along with any resource files and supporting libraries into an installation package (e.g. CAP, or JAR file [9,10]) that can be downloaded to a Java Card. On the Java Card, the on-card bytecode verifier would analyse the downloaded application and validate that it conforms to the Java language semantics.

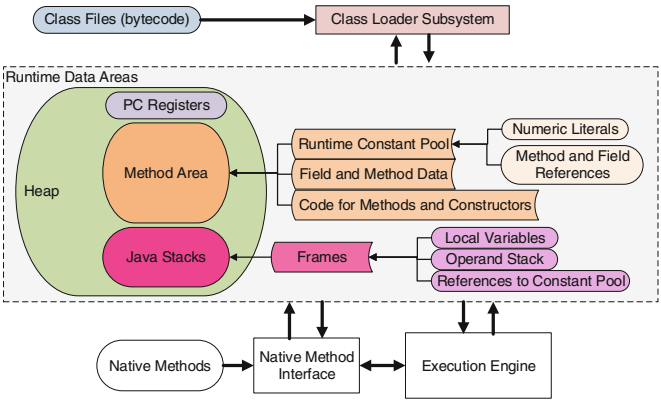

Fig. 1. Architecture of the Java card virtual machine

Figure 1 illustrates the architecture of a typical JCVM. Various components and their functions are described subsequently with emphasis on how they interact during the execution of an application.

The JCVM mainly deals with an abstract storage unit called *word* that is the smallest storage unit that it can process. The actual size of a *word* is left to the JCVM implementers. However, the JCVM specification [9] states that a word should be large enough to hold a value of `byte`, `short`, `reference`, or `returnAddress`.

When an application is initiated, the bytecode representation of an application is loaded into the JCVM memory by a "class loader subsystem". The class loader is responsible for locating and loading the class onto the memory areas used by the JCVM. This memory is divided into sub-areas, where each of them contains specific information regarding the application. The JCVM memory area is the heap, and all data/code related to an application is loaded onto it. The three main storage structures defined on the heap that are of relevance here are the Program Counter (PC) registers, method area, and Java stacks. These storage structures are briefly discussed here as they are referred to in the remaining paper (i.e., when we discuss our proposed counter-measures).

The PC registers store the memory address of the bytecode instruction currently executing. If the JCVM supports multiple threading then each thread will have its own PC register.

The method area is a memory space that consists of structures that include runtime constant pool, field and method data, and code related to methods and constructors. The runtime constant pool stores the constant field values (e.g. numeric literals) and references to the memory address related to methods and fields. The other two structures store the data and code related to fields and methods, etc.

A frame is created by the JCVM each time a method is invoked during the execution of an application. A frame is a construct that stores data, partial results, return values, and dynamically resolved links, associated with a single method (not the related class). These frames are stored on a last-in first-out (LIFO) stack called Java Stack. For each thread, there will be a different Java Stack. For security reasons, only the JCVM can issue the push and pop instructions to Java Stacks. The data structures that reside on a frame include an array of local variables, operand stack, and references to constant pool. The operand stack is a LIFO stack and it is empty when a frame is created. During the execution of a method, the JCVM will load data values (of either constant or non-constant variables/fields) onto the operand stack. The JCVM will operate on the values at the top of the operand stack and push the results back on it.

The JCVM provide well-defined interfaces to access native methods; however, contrary to traditional Java virtual machines they do not allow user-defined native methods. Each JCVM has an execution engine that is responsible for the execution of the individual instructions (opcodes) in an application code. The design of the execution engine is dependent on the underlying hardware platform and in a simple way, it can be considered as a software interface to the platform's processor.

This section does not exhaustively explain the JCRE and the rationale for covering the aforementioned topics is to make it easy to follow the subsequent discussion in the paper.

2.2 Related Work on JCRE Security

Earlier work on Java Cards was mainly related to the semantic and formal modelling of the JCVM [11,12], Java Card firewall mechanism [13,14], and applets [15–17]. The JCRE countermeasure against ill-formed applications was based on on-card bytecode verification [18–21], which became compulsory in the Java Card version 3 [9].

In the early 2000s, side channel analysis and fault attacks on smart card platforms were mainly focussed on the cryptographic algorithms [2,22–26]. However, in recent years, logical and fault attacks are combined to target the JCRE [27–29].

In 2008, Mostowski and Poll [30] loaded an ill-typed bytecode on various smart cards to test their security and reliability mechanisms. They also noted that smart cards that had an effective on-card bytecode verifier were less susceptible than others. In 2009, Hogenboom and Mostowski [31] managed to read

arbitrary contents of the memory. They performed this attack even in the presence of the Java Card firewall mechanism. Similar results were also shown by Lanet and Iguchi-Cartigny [32]. Sere et al. [33] use the similar attack of modifying the bytecodes to gain unauthorised access or skip the security mechanism on a platform. However, Sere et al. relied on fault attacks to modify the bytecodes rather than modifying them off-card as done by [30]. This way, Sere et al. managed to bypass the on-card bytecode verification. A countermeasure to this attack provided by Sere et al. relied on tagging the bytecode instructions with integrity values (i.e. integrity bits) and during the execution, the JCVM checks these bits and if it fails, the execution terminates.

In 2010, Barbu et al. [7] along with Vétillard and Ferrari [6] used a similar attack methodology to Sere et al. [33] that later came to be known as combined attacks. Later, the combined attack technique was extended to target various components of JCVM in [34–36]. These attacks are significant; nevertheless, they require the loading of an application designed specifically to accomplish the attack goals.

Dubrile et al. [48] discussed the fault enabled mutants in the Java Cards and proposed a countermeasure based on the typed stack. In [49] Julien Lancia illustrated a combined attack on the memory references (object and variable references) and proposed a countermeasure based on a defensive virtual machine.

The discussion in this section is by no means exhaustive but it introduces the challenges faced by the JCRE.

2.3 Motivation

During an application's lifetime, the application's security is dependent on the security of the runtime environment. As discussed in Sect. 2.2, a smart card runtime environment is increasingly facing the convergence of various attack techniques (e.g. fault and logical attacks). Although, physical protection mechanisms regarding fault attacks are proposed [37], we consider that the necessary software protection for the runtime environment cannot be understated. The software protection can augment the hardware mechanism to protect against the combined attacks, as a similar approach has yielded successful results in the secure design of cryptographic algorithms for smart cards [38]. Therefore, in this paper, we will focus on the software protection mechanism.

In the literature, several methods are described for software protection mechanism, including application slicing in which an application is partitioned for performance [39] or to protect intellectual property [40]. Such partitioning can be used to tag individual segments of an application with adequate security requirements. The runtime environment can then take into account the security requirements, tagged with individual segments during the execution; thus providing configurable runtime security architecture. A similar approach is proposed by Java Card 3 [9] and as part of the counter-measures to combined attacks proposed by Sere et al. [36] and Bouffard et al. [41]. These proposals are based on using Java annotations to tag segments of an application with required security or reliability levels.

Developers can use Java annotations to provide information regarding an application (or its segment), which is used by either the compiler, or runtime environment (i.e. JCVM). Based on Java annotations, Bouffard et al. [41] and Sere et al. [36] proposed mechanisms to prevent control flow attacks. In addition, Loining et al. [42] used the Java annotations to ensure a secure and reliable development of applications for embedded devices (e.g. smart cards). Furthermore, Java Card 3 Connected Edition also makes provision for Java annotations [9]. The defined annotations by Java Card 3 are integrity, confidentiality, and full (which corresponds to both integrity and confidentiality). In addition, the specification also allows proprietary annotations that can be used to invoke specific protection mechanisms implemented by the respective card manufacturer. The Java Card 3 specification does not detail what operations a JCVM should perform when encountering a particular annotation, which are left to the discretion of the card manufacturers.

These proposals are useful but a malicious user can use the annotations to his advantage in order to accomplish malicious goals. To avoid this, in our proposal we have an on-card analyser that checks the security and reliability requirements of an application, validates the associated Java annotations (tags) with each segment of the application, and modifies the security annotations where adequate. In such a scenario, we may assume that tagging segments of an application with security annotations might be useful. Nevertheless, such an on-card analyser is not currently available on smart cards. In this paper, we solely focus on adequately hardening the runtime environment.

In our proposed framework, we tackle the problem from three aspects: application compilation, runtime protection, and trusted component. The Java annotations are used to tag properties of individual segments of an application. Runtime commands (opcodes) that might be subverted to gain unauthorised access are hardened with additional protection (security checks), and finally a trusted component is included to complement the runtime environment for security verification and validation of an application's execution.

3 Runtime Protection Mechanism

In this section, we describe the anticipated attacker's capability along with the security requirements for a reliable and safe JCRE. Subsequently, we discuss the proposed runtime protection mechanism and how it provides a secure and reliable framework for JCRE.

3.1 Attacker's Capability

Due to the advancement in chip technology and hardware protection mechanisms [43], we have taken a realistic approach in defining the attacker's capability, taking into consideration the current state-of-the-art in attack methodologies for smart cards. The attacker's capabilities are listed as below:

1. Has the knowledge of the underlying (hardware and software) architecture.
2. Has the ability to load a customised application onto a given smart card.

3. Has the capability to induce a fault attack at a precise clock cycle.
4. Has the limited capability of changing a byte value to either 0x00 or 0xFF, or a random value in between.
5. Has the potential to change values stored in a non-volatile memory permanently within the limits of the capability four.
6. Has the ability to inject multiple faults; however, only in serial fashion (i.e. after injecting a fault, the attacker waits for the results before injecting the next fault). The adversary cannot inject multiple faults in parallel — injecting two faults simultaneously.
7. Can overwrite the whole or part of a memory such as the Electrically Erasable Programmable Read-Only Memory (EEPROM) or off-card storage.

Capability four restricts an adversary to induce a precise byte error rather than the precise bit error. This restriction is based on the underlying smart card hardware architecture. This is not to say that precise bit errors are not possible in smart cards. On the contrary, they are technically possible but increasing the density of packaging (i.e. chip fabrication) makes it challenging to change a value of a bit in comparison to changing the value of a byte.

The rationale behind the choice of multiple fault attacks in serial fashion than parallel is to give precise control and reproducibility of the attack. In fault attacks where a malicious user injects multiple faults simultaneously (parallel), it is difficult to assess whether the first fault injection was successful; therefore, injecting the second fault may be less productive.

3.2 Security Requirements for a Runtime Protection Mechanism

In this section, we discuss the set of requirements appropriate for a runtime protection mechanism to defend against the attacker discussed in the previous section.

1. Customisable: Enables the application developers to define the security requirements (if preferred) for their applications, which will be enforced as long as they do not violate the platform's and/or other application's security requirements.
2. Developer Independent: Does not require the application developers to evaluate the security risks of their application and adequately tag it.
3. Code Integrity: Detect any unauthorised modification to the application code before it is executed.
4. Stack Integrity: Detect any modification to the values stored on the Java stacks (e.g. operand stack).
5. Execution Flow Evaluation: Detect any illegal jumps to either restricted areas (e.g. data or code locations for a different application) or violating the secure execution flow of the application.

These requirements are revisited in Sect. 4.5, when our proposal is compared with the existing proposals discussed in Sect. 2.2.

3.3 Overview of the Proposed Runtime Protection Mechanism

The proposed architecture of the runtime protection mechanism is involved at various stages of the application lifecycle - including the application compilation, on-card bytecode verification, and execution as shown in Fig. 2.

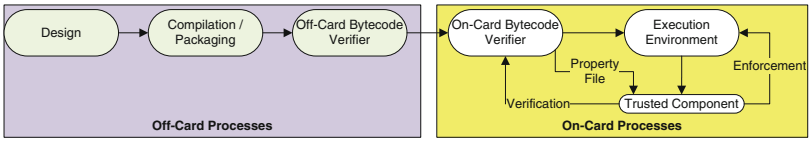

Fig. 2. Generic overview of the proposed runtime protection mechanism

During the compilation/packaging process additional information regarding individual methods, classes, and objects of an application is generated as part of the property file, discussed in Sect. 3.4. The property file assists the runtime environment to provide a security and reliability service during the execution of the application. The off-card bytecode verification checks whether the downloaded application conforms to the (given) language's semantics. The on-card bytecode verifier can also request the trusted component to validate the property file. The trusted component is the proposed Runtime Security Manager (RSM) discussed in Sect. 3.6 that actively enforces the security and reliability policy of the platform - taking into account the information included in the property file.

The proposed framework does not require that application developers perform security assessments of their application(s) to adequately tag application segments. The framework only requires at minimum that developers compile their applications in a way that they have property files that stores information related to the respective applications. The second requirement of the proposed framework is to adequately harden the runtime environment discussed in Sect. 3.5 along with introducing the RSM that will enforce the platform security policy (Sect. 3.6).

In subsequent sections, we will extend the generic architecture discussed in this section and explain how the different components come together.

3.4 Application Compilation

A Java compiler will take a Java file and convert it to a (bytecode) class file. The class file not only has opcodes, but it also includes information about various segments (e.g. methods, and classes) of an application that is necessary for the JCVM to execute the application. However, for our proposal we introduce a property file that includes additional information about an application. If a JCVM knows how to process property files then it will proceed with them; otherwise, it will ignore them. In our proposal a property file is stored and used by the RSM during the execution of the associated application. In order

to integrate the RSM into the runtime environment, the JCVM is required to be modified so it can communicate with the RSM in order to safeguard the execution environment.

```
1  ApplicationInfo{
2        Application_Identifier ApplicationIdentifier;
3        ClassInformation ClassInfo[class_count];}
4  ClassInfo{
5        Class_Identifier ClassIdentifier;
6        MethodInformation MethodInfo[method_count];}
7  MethodInfo{
8        Method_Identifier MethodIdentifier;
9        MethodIntegrity HashValue;
10       PermittedExecutionPath Path[jumps_count]}
```

Listing 1.1. Property file structure of a Java Card application.

The property file contains security and reliability information concerning an application that the runtime environment can utilise to execute an application. The structure of the property file is illustrated in Listing 1.1, which includes information regarding the permitted execution-paths, and integrity matrix (hash values of the non-mutable part of the individual methods in a class).

The `ApplicationInfo` data structure includes the application identifier (e.g. AID) and an array of classes that are part of the respective application. For each class in the application, we have a `ClassInfo` structure that contains the `MethodInformation` array that contains information regarding all methods associated with the given class. Each method is represented by the `MethodInfo` structure that includes the permitted execution-paths that are generated for each method. In the permitted execution-paths, child nodes represent jumps to other methods, irrespective of whether they are from the same application or from a different application. In a way, combining the method paths of all classes can give the complete permitted execution-path of the respective application. In addition to the permitted execution-paths, a `MethodInfo` also contains the hash value (of non-mutable code) of the respective method. This hash value can be generated at compile time and added to the property file, or at the time of the application installation: the RSM calculates the hash value and stores it in the property file.

3.5 Execution Environment

The runtime environment is modified to support the inclusion of the RSM that is shown in Fig. 3. At the time of application installation, the application bytecode is stored in the respective SP's domain along with the associated property file. The property file is sealed[1] so that neither the application nor an off-card entity (e.g. an SP or/and adversary) can modify it without detection. At the time of execution, the RSM will retrieve the file, verify the integrity of the file, and then decrypt it. If an SP wants to update its application then it will proceed with the

[1] Sealed: The data is encrypted (authenticated encryption) by the RSM storage key.

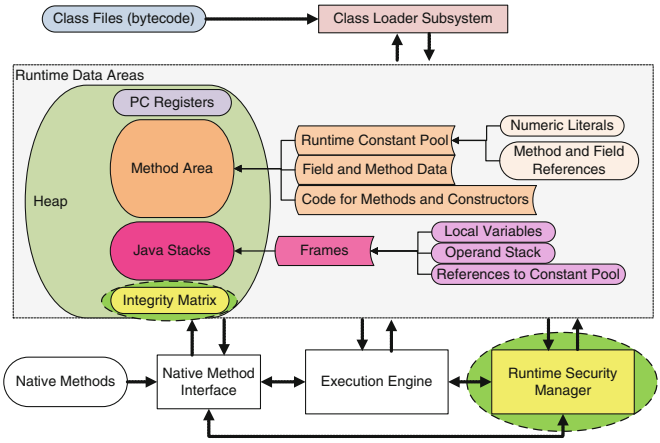

Fig. 3. Architecture of the proposed runtime environment for COM devices.

update command[2] that will notify the RSM of the update. At the completion of the update, the RSM will verify the application security certificate (if available), and update the property file – if required.

3.6 Runtime Security Manager

The purpose of the RSM is to enforce the security counter-measures (Sect. 3.7) defined by the respective platform. To enforce the security counter-measures, the RSM has access to the heap area (e.g. method area, Java stacks) and it can be implemented as either a serial or a parallel mode.

A serial RSM will rely on the execution engine of the JCVM (Fig. 1) to perform the required tasks. This means that when an execution engine encounters instructions that require an enforcement of the security policy, it will invoke the RSM that will then perform the checks. If successful the execution engine continues with execution, otherwise, it will terminate. A parallel RSM will have its own dedicated hardware (i.e. processor) support that enables it to perform checks simultaneously while the execution engine is executing an application. Note that having multiple processors on a smart card is technically possible [44]. The main question regarding the choice is not the hardware, but the balance between the performance and latency.

Performance, as the name suggests is concerned with the computational speed. Whereas, latency in this context deals with the number of instructions executed between an injected-error to the point it is detected. We will return to this discussion later in Sect. 4 where we provide test (simulated) implementation results.

[2] Update Command: We do not propose any update command in this paper but similar commands are defined as part of the GlobalPlatform card specification. The update command enables an authorised entity (e.g. SP) to modify an application.

3.7 Runtime Security Counter-Measures

The RSM along with the runtime environment would apply the required security counter-measures (as part of the runtime protection mechanism) that are discussed in subsequent sections.

Operand Stack Integrity. As discussed in Sect. 2.1, an operand stack is part of the Java stacks and they are associated with individual Java frames (methods). During the execution of an application, the runtime environment pushes and pops local variables, constant fields, and object references to the operand stack. The instructions specified in an application can then process the values at the top of the stack. Barbu et el. [34] showed that a fault injection that changes the values stored on the operand stack could have adverse effect on an application's security. Furthermore, they also provided three different counter-measures to the proposed attack.

The proposed countermeasure (second-refined method) of Barbu et al. [34] is based on the idea of operand stack integrity. They define a variable α, and all values that are pushed on or popped from the operand stack are XORed with the α. On every jump instruction beyond the scope of the current frame (method), the runtime environment XORs all the values stored on the operand and compares the result with α. If they match then the integrity of the operand stack is verified. Their proposal does not measure the integrity of the operand stack on instructions like if-else or loops, which could be the target of the malicious user. In their proposed counter-measures they sacrificed security and (to some extent) performance for the sake of memory use, whereas our proposal focuses on security rather than saving the memory.

In our proposal, we use a Last In First Out (LIFO) stack referred to as integrity stack. One thing to note is that the JCVM knows the size of the operand stack when it loads a frame (Sect. 2.1); therefore, the RSM just creates an integrity stack of the size n where n is the size of the respective operand stack. We refer to the integrity stack as "InS" in Listing 1.2.

When a frame is loaded, the JCVM and the RSM will create an operand and integrity stack, respectively. Furthermore, the RSM will also generate a random number and stores it as S_r. The rationale for using the random number will become apparent in the subsequent discussion.

```
1 // Executed by RSM when a value is pushed onto an integrity stack.
2 On_Stack_Push(pushedValue){
3  push(InS[top] XOR pushedValue);}
4 // Executed by RSM when a value is popped from an operand stack.
5 On_Stack_Pop(poppedValue){
6    if(pop(InS) XOR poppedValue := InS[top]){
7    }else{
8     terminateExecution();
9     }}
```

Listing 1.2. Operand stack integrity operations.

When a value V_i is pushed to the operand stack, if it is the first value on the stack then the value pushed on the InS will be $I_i = V_i \oplus S_r$. For all subsequent values (where $i > 1$) the values pushed on the Ins will be $I_i = V_i \oplus V_{i-1}$.

The rationale for using a random number is to avoid parallel fault injections that try to change the values on both operand and integrity stack simultaneously. Such a parallel fault injection will become difficult if an adversary cannot predict the values stored on the integrity stack, as each value on the integrity stack will be chained with the generated random number.

When a value is popped out of the operand stack, we also pop the integrity value from the integrity stack, XOR it with the popped value from the operand stack and compare it with the new top value on the integrity stack. If the values match then the integrity of the popped value from the operand stack is verified; otherwise, it has been corrupted and the RSM requests the JCVM to terminate the execution as shown in Listing 1.2.

The RSM will continuously monitor the integrity of the operand stack, in comparison to the Barbu's proposal. Furthermore, in this proposal the validation does not require the calculation of integrity value over the entire operand stack. If we take the Barbu's proposal then for an operand stack of length 'n', we have to perform "n-1" XOR operations every time we need to verify the state of the operand stack. However, in our proposal we only need to perform one XOR operation. We sacrifice the memory for the sake of performance in our proposal. We consider that operand stacks are not large data structures so even if we double the memory used by them, it will not have an adverse effect on the overall memory usage.

Permitted Execution Path (PEP) Analysis. In our proposal, we are concerned with jumps that refer to external resources. The term external resources in the context of PEP analysis means any jump that goes beyond the scope of the current Java frame (i.e. method) while it is still on the Java stack. Once a method completes its execution, the JCVM will remove the associated Java frame from the Java stacks (Fig. 1). Examples of such jumps defined in Java virtual machine specification [45] are `invokeinterface`, `invokestatic`, `invokevirtual`, `areturn`, etc.

```
1 byte B(byte inputValue){
2   byte a = 1;
3   if(inputValue != a){ C(inputValue);
4   }else{D(inputValue)}
5   return SG(inputValue);}
```

Listing 1.3. Code for an example method B.

To explain the Permitted Execution Path Analysis further, we consider an example method `B` that has three jumps before it reaches the return statement that completes the execution of the method. The method `B`'s code is shown in Listing 1.3. Each invocation of a method (e.g. `C`, `D`, and `SG`) is represented by a symbolic method name (i.e. alphanumeric form that is easily readable/

recognisable by humans) that has an associated unique byte sequence referred as method identifier. For example, unique method identifier of methods `B`, `C`, `D`, and `SG` are 0xF122, 0xF123, 0xF124, and 0xF125, respectively. For explanation we have used method identifiers that consist of two bytes. Along with the method identifier the property file also includes `PermittedExecutionPath`, which is a set of PEPs sanctioned for the given method.

The `PermitedExecutionPath` in the property file (Listing 1.1) is simply constructed by taking into account every possible (legal) execution path of a method. Taking the example method `B`, the first jump can either be to method `C` or `D` depending upon the input. The construction of the `PermitedExecutionPath` (set of legal jumps) is constructed by XORing the method identifiers of individual jumps.

The PEP analysis requires that the RSM have a PEP variable "cfa" that stores the path taken by an application as $cfa = \Sigma_{j=1}^{n} C_j$. Where C_j represents the jumps taken during execution of an application. During the execution of a method, when the JCVM encounters a jump to another method the RSM XORs the method identifier with the current value of "cfa" and lookup the `PermitedExecutionPath` of the given method in the associated property file. If it finds a matching value, the JCVM will proceed with the execution; if not it will terminate the execution. Our scheme also deals with the loop instructions that contain jumps to multiple methods depending upon the loop condition.

Bytecode Integrity. The property file associated with an application stores the hash values of individual methods. When the runtime environment fetches an application, the RSM will measure the integrity value of individual methods of the application and compare them with the hash values in the property file. Therefore, any method that is loaded to the heap goes through the integrity validation. This validation protects against the fault attacks on an application stored while it is stored on a non-volatile memory.

The hashes of the individual methods (code and constant local-data variables) along with the integrity values (hash values) generated on the global persistent data can create a whole application integrity matrix. During the execution of an application, when it jumps from one method to another, it can be assured that the execution path is going to a (potentially) trusted method or the integrity of the called method will be verified. The RSM also tracks the integrity of the global variables and update the hash values if any authorised changes are being performed by the application.

4 Analysis of the Runtime Protection Mechanism

In this section, we evaluate the suitability of countermeasures against the attacks discussed in Sect. 2.2 under the adversary's capability detailed in Sect. 3.1. Furthermore, we provide the latency and incurred overhead analysis for both serial and parallel RSMs.

4.1 Security Analysis

In this section, we discuss how the proposed counter-measures protect against the combined attacks under the attacker's capability detailed in Sect. 3.1.

Operand Stack Integrity. We proposed a more refined approach to Barbu et al. [34] and removed the need to perform integrity measurement of the entire operand stack on each validation. In addition, we made the validation process continuous thus checking the integrity of the operand stack on each pop and push operation. If a malicious user changes values on the operand stack, the RSM can not only detect the modification but can also provide error correction service by providing the correct value that was stored on the operand stand. Furthermore, by using a random number, our proposal makes it difficult for an adversary to know the values stored on the integrity stack, even if he has the knowledge of all values on the operand stack.

PEP Analysis. The PEP analysis performed by the RSM during the execution of an application effectively prevents execution path attacks. If an attacker has the capability of multiple fault injections simultaneously, (which is beyond the capability of our attacker as stated in Sect. 3.1) then he can in theory affect the RSM execution. Nevertheless, even with simultaneous injections the attacker may be able to skip a node in the execution tree but the RSM calculation on the subsequent nodes will reveal an illegal path of execution. Therefore, even in the parallel injection model the RSM will detect the erroneous execution path, unless the attacker keeps on injecting faults during the whole execution of an application.

Bytecode Integrity. Our countermeasure prevents an adversary from changing an application while it is stored on a non-volatile memory (capability four of an adversary discussed in Sect. 3.1). To avoid such modifications, the RSM generates a hash of individual methods that are requested by the JCVM. If the hash matches the stored value (`MethodIntegrity` in Listing 1.1) in the respective property file, the JCVM will proceed with the execution of the method; otherwise, the RSM will signal the termination of the application (and possibly mark it malicious and up for deletion). Furthermore, this protection mechanism can also safeguard the dynamic loading of applications/classes/routines as part of a simple web server or other applications, which are stored in off-card storage.

4.2 Evaluation Context

For evaluating the proposed counter-measures, we have selected four sample applications. Two of the applications selected are part of the Java Card development kit distribution: Wallet and Java Purse. The other two applications are the offline attestation algorithm [46] and the STCP_{SP} protocol [47].

4.3 Latency Analysis

As discussed before, latency is the number of instructions executed after an adversary mounts an attack, before the system becomes aware of it. Therefore, in this section we analyse the latency of the proposed counter-measures under the concepts of serial and parallel RSMs that are listed in Table 1 and discussed subsequently.

Table 1. Latency measurement of individual countermeasure

Counter-measures	Serial RSM	Parallel RSM
Operand Stack Integrity	0 + i	3 + i
Permitted Execution Path Analysis	0	$3(C_n)$
Bytecode Integrity	0	0

In case of the operand stack integrity, the serial RSM finds the occurrence of an error (e.g. fault injection) with latency "0+i", where 'i' is the number of instructions executed before the manipulated value reaches the top of the operand stack. Similarly, the latency value in case of the operand stack integrity for the parallel RSM is "3+i", where '3' is the number of instructions required to perform a comparison on pop operation (`On_Stack_Pop(poppedValue)` in Listing 1.2). The latency value of the parallel RSM is higher than the serial. This has to do with the fact that while the parallel RSM is applying the security checks the JCVM does not need to stop the execution of subsequent instructions.

Regarding the PEP analysis, the serial RSM has a latency of zero where the parallel RSM has latency value of "$3(C_n)$", where the value C_n represents the number of legal jumps in the respective `PermittedExecutionPath` set. To explain this further, consider the example in Listing 1.3. The set method `B` has four possible values ($B_{cfa-Set}$ in Sect. 3.7). Thereby, the latency value for a jump in the method `B` in the worse case is "3(4) = 12". The value '3' represents the number of instructions required to execute individual comparison.

A notable point to mention here is that all latency measurements listed in Table 2 are based on the worst-case conditions. Furthermore, it is apparent that it might be difficult to implement a complete parallel RSM. To explain our point, consider two consecutive jump instructions in which the parallel RSM has to perform PEP analysis. In such a situation, there might be a possibility that while the RSM is still evaluating the first jump, the JCVM initiates the second jump instruction. Therefore, this might create a deadlock between the JCVM and parallel RSM, so we consider that either JCVM should wait for the RSM to complete the verification, or for the sake of performance the RSM might skip certain verifications. We opt for the parallel RSM that will switch to the serial RSM mode, restricting the JCVM to proceed with next instruction until the RSM can apply the security checks. This situation will be further explained during the discussion on the performance measurements in the next section.

4.4 Incurred Overhead Analysis

To evaluate the performance impact of the proposed counter-measures we developed an abstract virtual machine that takes the bytecode of each Java Card applet and then computes the computational overhead for individual counter-measure. When a Java application is compiled, the Java compiler (`javac`) produces a class file as discussed in Sect. 2.1. The class file is a Java bytecode representation, and we utilise the `javap` tool that comes with Java Development Kit (JDK) as it produces the bytecode representation of a class file in human-readable mnemonics as represented in the JVM specification [45]. The abstract virtual machine takes the mnenomic bytecode representation and searches for push, pop, and jump (e.g. method invocation) opcodes. Subsequently, we calculated the number of extra instructions required to be executed in order to implement the counter-measures discussed in previous sections.

Table 2. Performance measurement (percentage increase in computational cost)

Applications	Serial RSM	Parallel RSM
Wallet	+29 %	+22 %
Java Purse	+30 %	+26 %
Offline Attestation [46]	+27 %	+23 %
STCP_{SP} [47]	+39 %	+33 %

We compute the incurred overhead as a number of instructions that are going to be executed by an application that our countermeasures verify/validate. After this measurement, we have associated costs based on additional instructions executed for each JCVM instruction and calculated as an (approximate) increase in the percentage of computational overhead and listed in Table 2. Furthermore, the computational cost of generating a hash is dependent on individual hardware configuration. The same is also true for the execution of each of the instructions. This is the reason why we opt for the evaluation based on the number of increased instructions rather then the performance as it provides us a hardware agnostic cost.

For each application, the counter-measures have different computational overhead values because they depend upon how many times certain instructions that invoke the counter-measures are executed. Therefore, the computational overhead measurements in Table 2 can only give us a measure of how the performance is affected in individual cases - without generalising for other applications.

4.5 Comparative Analysis

In this section, we will compare our proposed framework with the existing state-of-the-art discussed in Sect. 2.2 in the context of security requirements for a JCRE (listed in Sect. 3.2).

Table 3. Comparison between Proposed and Existing Proposals

Requirement	Barbu et al. [7]	Sere et al. [36]	Dubreuil et al. [48]	Lancia [49]	RSM
Customisable	Limited	Yes	No	No	Yes
Developer Independent	Yes	No	Yes	Yes	Yes
Code Integrity	No	Yes	No	Yes	Yes
Stack Integrity	Yes	No	Yes	No	Yes
Execution Flow Evaluation	No	No	No	No	Yes

Table 3 illustrates our proposed framework's results are better in comparison to other proposals, in the context of attacker capabilities and security requirements. One thing to note is that other proposals compared in Table 3 do not provide performance degradation matrix associated with their countermeasures, equivalent to one presented in Sects. 4.3 and 4.4

5 Conclusion

In this paper we discussed the smart card runtime environment by taking the Java Card as a practical example. The JCRE was described with its different data structures that it uses during the execution of an application. Subsequently, we discussed various attacks that target the smart card runtime environment and most of these attacks are based on perturbation of the values stored by the runtime environment. These perturbations are the result of fault injection, which was defined and mapped to an adversary's capability in this paper. Based on recent published attacks on the smart card runtime environment, we proposed an architecture that includes the provision of a RSM. We also proposed three counter-measures and provided the computational cost imposed by them. The overall protection framework is then compared with the existing frameworks and we showed that our proposal provides comparatively better protection. No doubt, counter-measures that do not change the core architecture the Java virtual machine, will almost always incur extra computational cost. Therefore, we concluded in this paper that a better way forward would be to change the architecture of the Java virtual machine. Finally, in the context of this paper we showed that the current architecture can be hardened at the expanse of a modest computational penalty.

References

1. Anderson, R., Kuhn, M.: Low cost attacks on tamper resistant devices. In: Christianson, B., Lomas, M., Crispo, B., Roe, M. (eds.) Security Protocols 1997. LNCS, vol. 1361, pp. 125–136. Springer, Heidelberg (1998)
2. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
3. Sauveron, D.: Multiapplication smart card: towards an open smart card? Inf. Secur. Tech. Rep. **14**(2), 70–78 (2009)

4. Akram, R.N., Markantonakis, K.: Smart cards: state-of-the-art to future directions, invited paper. In: Douligeris, C., Serpanos, D. (eds.) IEEE International Symposium on Signal Processing and Information Technology (ISSPIT 2013). IEEE CS, Athens, Greece (2013)
5. Markantonakis, K., Mayes, K., Sauveron, D., Askoxylakis, I.: Overview of security threats for smart cards in the public transport industry. In: 2008 IEEE International Conference on e-Business Engineering. IEEE CS (2008)
6. Vétillard, E., Ferrari, A.: Combined attacks and countermeasures. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 133–147. Springer, Heidelberg (2010)
7. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java card 3.0 combining fault and logical attacks. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 148–163. Springer, Heidelberg (2010)
8. Chaumette, S., Sauveron, D.: An efficient and simple way to test the security of Java cards. In: Fernández-Medina, E., Castro, J.C.H., Castro, L.J.G. (eds.) Security in Information Systems, pp. 331–341. INSTICC Press, Miami (2005)
9. Java Card Platform Specification, Oracle Std. v3.0.1, May 2009
10. Java Card Platform Specification, Sun Microsystem Inc Std. v2.2.2, March 2006
11. Barthe, G., Dufay, G., Jakubiec, L., de Sousa, S.M.: A formal correspondence between offensive and defensive JavaCard virtual machines. In: Cortesi, A. (ed.) VMCAI 2002. LNCS, vol. 2294, pp. 32–45. Springer, Heidelberg (2002)
12. Barthe, G., Stratulat, S.: Validation of the JavaCard platform with implicit induction techniques. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 337–351. Springer, Heidelberg (2003)
13. Éluard, M., Jensen, T., Denne, E.: An operational semantics of the Java card firewall. In: Attali, S., Jensen, T. (eds.) E-smart 2001. LNCS, vol. 2140, pp. 95–110. Springer, Heidelberg (2001)
14. Éluard, M., Jensen, T.: Secure object flow analysis for Java card. In: Proceedings of the 5th Conference on Smart Card Research and Advanced Application Conference, CARDIS 2002, p. 11. USENIX Association, California (2002)
15. Lanet, J.L., Requet, A.: Formal proof of smart card applets correctness. In: Schneier, B., Quisquater, J.-J. (eds.) CARDIS 1998. LNCS, vol. 1820, pp. 85–97. Springer, Heidelberg (2000)
16. Meijer, H., Poll, E.: Towards a full formal specification of the JavaCard API. In: Attali, S., Jensen, T. (eds.) E-smart 2001. LNCS, vol. 2140, pp. 165–178. Springer, Heidelberg (2001)
17. Almaliotis, V., Loizidis, A., Katsaros, P., Louridas, P., Spinellis, D.D.: Static program analysis for Java card applets. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 17–31. Springer, Heidelberg (2008)
18. Basin, D., Friedrich, S., Posegga, J., Vogt, H.: Java bytecode verification by model checking. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 491–494. Springer, Heidelberg (1999)
19. Leroy, X.: On-card bytecode verification for Java card. In: Attali, S., Jensen, T. (eds.) E-smart 2001. LNCS, vol. 2140, pp. 150–164. Springer, Heidelberg (2001)
20. Basin, D., Friedrich, S., Gawkowski, M.: Verified bytecode model checkers. In: Carreño, V.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2002. LNCS, vol. 2410, pp. 47–66. Springer, Heidelberg (2002)
21. Leroy, X.: Bytecode verification on Java smart cards. Softw. Pract. Exper. **32**(4), 319–340 (2002)

22. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Kaliski Jr, B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 513–525. Springer, Heidelberg (1997)
23. Messerges, T.S., Dabbish, E.A., Sloan, R.H.: Investigations of power analysis attacks on smartcards. In: Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology, p. 17. USENIX Association, Berkeley (1999)
24. Skorobogatov, S.P., Anderson, R.J.: Optical fault induction attacks. In: Kaliski Jr, B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 2–12. Springer, Heidelberg (2003)
25. Quisquater, J.-J., Samyde, D.: Eddy current for Magnetic Analysis with Active Sensor. Springer (2002)
26. Aumller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.-P.: Fault attacks on RSA with CRT: concrete results and practical countermeasures. In: Kaliski Jr, B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 260–275. Springer, Heidelberg (2003)
27. Joint Interpretation Library - Application of Attack Potential to Smartcards, Online, Technical report, Apirl 2006
28. Vertanen, O.: Java type confusion and fault attacks. In: Breveglieri, L., Koren, I., Naccache, D., Seifert, J.-P. (eds.) FDTC 2006. LNCS, vol. 4236, pp. 237–251. Springer, Heidelberg (2006)
29. Lemarechal, A.: Introduction to fault attacks on smartcard. In: 11th IEEE International On-Line Testing Symposium, IOLTS 2005, p. 116, July 2005
30. Mostowski, W., Poll, E.: Malicious code on Java card smartcards: attacks and countermeasures. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 1–16. Springer, Heidelberg (2008)
31. Hogenboom, J., Mostowski, W.: Full memory read attack on a Java card. In: Pereira, O., Quisquater, J.-J., Standaert, F.-X. (eds.) 4th Benelux Workshop on Information and System Security. Springer, Belgium (2009)
32. Lanet, J.-L., Iguchi-Cartigny, J.: Developing a Trojan applet in a smart card. J. Comput. Virol. 6(1) (2009)
33. Sere, A.A., Iguchi-Cartigny, J., Lanet, J.-L.: Automatic detection of fault attack and countermeasures. In: Proceedings of the 4th Workshop on Embedded Systems Security, ser. WESS 2009, pp. 71–77. ACM, New York (2009)
34. Barbu, G., Duc, G., Hoogvorst, P.: Java card operand stack: fault attacks, combined attacks and countermeasures. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 297–313. Springer, Heidelberg (2011)
35. Barbu, G., Thiebeauld, H.: Synchronized attacks on multithreaded systems - application to Java card 3.0 -. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 18–33. Springer, Heidelberg (2011)
36. Sere, A.A., Iguchi-Cartigny, J., Lanet, J.-L.: Evaluation of countermeasures against fault attacks on smart cards. Int. J. Secur. Appl. **5**(2), 49–61 (2011)
37. Derouet, O.: Secure smartcard design againist laser fault. (Invited Speaker). In: 4th Workshop on Fault Diagnosis and Tolerance in Cryptography (FDRC 2007). IEEE-CS, Austria, Vienna, September 2007
38. Kim, S.-K., Kim, T.H., Han, D.-G., Hong, S.: An efficient CRT-RSA algorithm secure against power and fault attacks. J. Syst. Softw. **84**(10), 1660–1669 (2011)
39. Zhang, T., Pande, S., Valverde, A.: Tamper-resistant whole program partitioning. In: LCTES 2003, the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems, pp. 209–219. ACM, New York (2003)

40. Zhuang, X., Zhang, T., Lee, H.-H.S., Pande, S.: Hardware assisted control flow obfuscation for embedded processors. In: CASES 2004. ACM, USA (2004)
41. Bouffard, G., Lanet, J.-L., Machemie, J.-B., Poichotte, J.-Y., Wary, J.-P.: Evaluation of the ability to transform SIM applications into hostile applications. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 1–17. Springer, Heidelberg (2011)
42. Loinig, J., Steger, C., Weiss, R., Haselsteiner, E.: Identification and Verification of Security Relevant Functions in Embedded Systems Based on Source Code Annotations and Assertions. In: Samarati, P., Tunstall, M., Posegga, J., Markantonakis, K., Sauveron, D. (eds.) WISTP 2010. LNCS, vol. 6033, pp. 316–323. Springer, Heidelberg (2010)
43. Séré, A.A.K., Iguchi-Cartigny, J., Lanet, J.-L.: Checking the paths to identify mutant application on embedded systems. In: Kim, T., Lee, Y., Kang, B.-H., Slezak, D. (eds.) FGIT 2010. LNCS, vol. 6485, pp. 459–468. Springer, Heidelberg (2010)
44. Rankl, W., Effing, W.: Smart Card Handbook, 3rd edn. Wiley, New York (2003)
45. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley Longman, Amsterdam (1999)
46. Akram, R.N., Markantonakis, K., Mayes, K.: Remote attestation mechanism for user centric smart cards using pseudorandom number generators. In: Qing, S., Zhou, J., Liu, D. (eds.) ICICS 2013. LNCS, vol. 8233, pp. 151–166. Springer, Heidelberg (2013)
47. Akram, R.N., Markantonakis, K., Mayes, K.: A secure and trusted channel protocol for the user centric smart card ownership model. In: 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications. IEEE CS, Australia, July 2013
48. Dubreuil, J., Bouffard, G., Lanet, J., Cartigny, J.: Type classification against fault enabled mutant in Java based smart card. In: 2012 Seventh International Conference on Availability, Reliability and Security (ARES), August 2012
49. Lancia, J.: Java card combined attacks with localization-agnostic fault injection. In: Mangard, S. (ed.) CARDIS 2012. LNCS, vol. 7771, pp. 31–45. Springer, Heidelberg (2013)