A Defensive Virtual Machine Layer to Counteract Fault Attacks on Java Cards

Michael Lackner, Reinhard Berlach, Wolfgang Raschke,
Reinhold Weiss, and Christian Steger

Institute for Technical Informatics,
Graz University of Technology, Graz, Austria
{michael.lackner,reinhard.berlach,wolfgang.raschke,
rweiss,steger}@tugraz.at

Abstract. The objective of Java Cards is to protect security-critical code and data against a hostile environment. Adversaries perform fault attacks on these cards to change the control and data flow of the Java Card Virtual Machine. These attacks confuse the Java type system, jump to forbidden code or remove run-time security checks. This work introduces a novel security layer for a defensive Java Card Virtual Machine to counteract fault attacks. The advantages of this layer from the security and design perspectives of the virtual machine are demonstrated. In a case study, we demonstrate three implementations of the abstraction layer running on a Java Card prototype. Two implementations use software checks that are optimized for either memory consumption or execution speed. The third implementation accelerates the run-time verification process by using the dedicated hardware protection units of the Java Card.

Keywords: Java Card, Defensive Virtual Machine, Countermeasure, Fault Attack.

1 Introduction

A Java Card enables Java applets to run on a smart card. The primary purpose of using a Java Card is the write-once, run-everywhere approach and the ability of post-issuance installation of applets [21]. These cards are used in a wide range of applications (e.g., digital wallets and transport tickets) to store security-critical code, data and cryptographic keys. Currently, these cards are still very resource-constrained devices that include an 8- or 16-bit processor, 4kB of volatile memory and 128kB of non-volatile memory. To make a Java Card Virtual Machine run on such a constrained device, a subset of Java is used [19]. Furthermore, special Java Card security concepts, such as the Java Card firewall [18] and a verification process for every applet [15], were added. The Java Card firewall is a run-time security feature that protects an applet against illegal access from other applets. For every access to a field or method of an object, this check is performed. Unfortunately, the firewall security mechanism can be circumvented by applets

L. Cavallaro and D. Gollmann (Eds.): WISTP 2013, LNCS 7886, pp. 82–97, 2013.

that do not comply with the Java Card specification. Such applets are called malicious applets.

To counteract malicious applets, a bytecode verification process is performed. This verification is performed either on-card or off-card for every applet [15]. Note that this bytecode verification is a static process and not performed during applet execution. The reasons for this static approach are the high resource needs of the verification process and the hardware constraints of the Java Card. This behavior is now abused by adversaries. They upload a valid applet onto the card and perform a fault attack (FA) during applet execution. Adversaries are now able to create a malicious applet out of a valid one [5].

A favorite time for performing a FA is during the fetching process. At this time, the virtual machine (VM) reads the next Java bytecode values from the memory. An adversary that performs an FA at this time can change the readout values. The VM then decodes the malicious bytecodes and executes them, which leads to a change in the control and data flow of the applet. A valid applet is mutated by such an FA to a malicious applet [5,17,11] and gains unauthorized access to secret code and data [16,2].

To counteract an FA, a VM must perform run-time security checks to determine if the bytecode behaves correctly. In the literature, different countermeasures, such as control-flow checks [23], double checks [4], integrity checks [8] and method encryption [20], have been proposed. Barbu [3] proposed a dynamic attack countermeasure in which the VM executes either standard bytecodes or bytecodes with additional security checks.

All these works do not concentrate on the question of how these security mechanisms can be smoothly integrated into a Java Card VM. For this integration, we propose adding an additional security layer into the VM. This layer abstracts the access to internal VM resources and performs run-time security checks to counteract FAs. The primary contributions of this paper are the following:

- Introduction of a novel defensive VM (D-VM) layer to counteract FAs during run-time. Access to security-critical resources of the VM, such as the operand stack (OS), local variables (LV) and bytecode area (BA), is handled using this layer.
- Usage of the D-VM layer as a dynamic countermeasure. Based on the actual security level of the card, different implementations of the D-VM layer are used. For a low-security level, the D-VM implementation uses fewer checks than for a high-security level. The security level depends on the credibility of the currently executed applet and run-time information received by hardware or software modules.
- A case study of a defensive VM using three different D-VM layer implementations. The API of the D-VM layer is used by the Java Card VM to perform run-time checks on the currently executing bytecode.
- The defensive VMs are executed on a smart card prototype with specific HW security features to speed up the run-time verification process. The resulting run-time and main memory consumption of all implemented D-VM layers are presented.

Section 2 provides an overview of attacks on Java Cards and the current countermeasures against them. Section 3 describes the novel D-VM layer presented in this work and its integration into the Java Card design. Furthermore, the method by which the D-VM layer enables the concept of dynamic countermeasures is presented. Section 4 presents implementation details regarding how the three D-VM implementations are inserted into the smart card prototype. Section 5 analyzes the additional costs for the D-VM implementations based on the execution and main memory overhead. Finally, the conclusions and future work are discussed in Section 6.

2 Related Work

In this section, the basics of the Java Card VM and work related to FA on Java Cards are presented. Then, an analysis of work regarding methods of counteracting FAs and securing the VM are presented. Finally, an FA example is presented to demonstrate the danger posed by such run-time attacks for the security of Java Cards.

2.1 Java Card Virtual Machine

A Java Card VM is software that is executed on a microprocessor. The VM itself can be considered a virtual computer that executes Java applets stored in the data area of the physical microprocessor. To be able to execute Java applets, the VM uses internal data structures, such as the OS or the LV, to store interim results of logical and combinatorial operations. All of these internal data structures are general objects for adversaries that attack the Java Card [4,20,24].

For every method invocation performed by the VM, a new Java frame [19] is created. This frame is pushed to the Java stack and removed from it when the method returns. In most VM implementations, this frame internally consists of three primary parts. These parts have static sizes during the execution of a method. The first frame part is the OS on which most Java operations are performed. The OS is the source and destination for most of the Java bytecodes. The second part is the LV memory region. The LV are used in the same manner as the registers on a standard CPU. The third part is the frame data, which holds all additional information needed by the VM and Java Card Runtime Environment (JCRE) [18]. This additional information includes, for example, return addresses and pointers to internal VM-related data structures.

2.2 Attacks on Java Cards

Loading an applet that does not conform to the specification defined in [19] onto a Java Card is a well-known problem called a logical attack (LA). After an LA, different applets on the card are no longer protected by the so-called Java

sandbox model. Through this sandbox, an applet is protected from illegal write and read operations of other applets. To perform an LA, an adversary must know the secret key to install applets. This key is known for development cards, but it is highly protected for industrial cards and only known by authorized companies and authorities. In conclusion, LAs are no longer security threats for current Java Cards.

Side-channel analyses are used to gather information about the currently executing method or instructions by measuring how the card changes environment parameters (e.g., power consumption and electromagnetic emission) during run-time. Integrated circuits influence the environment around them but can also be influenced by the environment. This influence is abused by an FA to change the normal control and data flow of the integrated circuit. Such FAs include glitch attacks on the power supply and laser attacks on the cards [2,24]. By performing side-channel analyses and FAs in combination, it is possible to break cryptographic algorithms to receive secret data or keys [16].

In 2010, a new group of attacks called combined attacks (CA) was introduced. These CAs combine LAs and FAs to enable the execution of ill-formed code during run-time [5]. An example of a CA is the removal of the *checkcast* bytecode to cause type confusion during run-time. Then, an adversary is able to break the Java sandbox model and obtain access to secret data and code stored on the card [5,17]. In this work work, we concentrate on countering FAs during the execution of an applet using our D-VM layer.

2.3 Countermeasures against Java Card Attacks

Since approximately 2010, an increasing number of researchers have started concentrating on the question of what tasks must be performed to make a VM more robust against FAs and CAs. Several authors [22,8] suggest adding an additional security component to the Java Card applet. In this component, they store checksums calculated over basic blocks of bytecodes. These checksums are calculated off-card in a static process and added to a new component of the applet. During run-time, the checksum of executed bytecodes is calculated using software and compared with the stored checksums. If these checksums are not the same, a security exception is thrown.

Another FA countermeasure is the use of control-flow graph information [23]. To enable this approach, a control-flow graph over basic blocks is calculated off-card and stored in an additional applet component. During run-time, the current control-flow graph is calculated and compared with the stored control graph.

In [20], the authors propose storing a countermeasure flag in a new applet component to indicate whether the method is encrypted. They perform this encryption using a secret key and the Java program counter for the bytecode of every method. Through this encryption, they are able to counteract attacks that change the control-flow of an applet to execute illegal code or data.

Another countermeasure against FAs that target the data stored on the OS is presented in [4]. In this work, integrity checks are performed when data are pushed or popped onto the OS. Through this approach, the OS is protected against FAs that corrupt the OS data.

Another run-time check against FAs is proposed in [10,14], in which they create separate OSes for each of the two data types, *integralValue* and *reference*. With this approach of splitting the OS, it is possible to counteract type-confusion attacks. A drawback is that in both works, the applet must be preprocessed.

In [3], the authors propose a dynamic countermeasure to counteract FAs. Bytecodes are implemented in different versions inside the VM, a standard version and an advanced version that performs additional security checks. The VM is now able to switch during run-time from the standard to the advanced version. By using unused Java bytecodes, an applet programmer can explicitly call the advanced bytecode versions.

The drawbacks of current FA countermeasures are that most of them add an additional security component to the applet or rely on preprocessing of the applet. This has different drawbacks, such as increased applet size or compatibility problems for VMs that do not support these new applet components. In this work, we propose a D-VM layer that performs checks on the currently executing bytecode. These checks are performed based on a run-time policy and do not require an off-card preprocessing step or an additional applet component.

2.4 EMAN4 Attack: Jump Outside the Bytecode Area

In 2011, the run-time attack EMAN4 was found [6]. In this work a laser was used to manipulate the read out values from the EEPROM to 0x00. By this laser attack an adversary is able to change the Java bytecode of post-issuance installed applets during their execution.

The target time of the attack is when the VM fetches the operands of the *goto_w* bytecode from the EEPROM. Generally the *goto_w* bytecode is used to perform a jump operation inside a method. The *goto_w* bytecode consists of the operand byte 0xa8 and two offset bytes for the branch destination [19]. This branch offset is added to the actual Java program counter to determine the next executing bytecode. An adversary which changes this offset is able to manipulate the control flow of the applet.

With the help of the EMAN4 attack it is possible to jump with the Java program counter outside the applet bytecode area (BA), as illustrated in Figure 1. This is done by changing the offset parameters of the *goto_w* bytecode from 0xFF20 to 0x0020 during the fetch process of the VM. The jump destination address of the EMAN4 attack is a data array outside the bytecode area. This data array was previously filled with adversary defined data. After the laser attack the VM executes the values of the data array. This execution of adversary definable data leads to considerably more critical security problems, such as memory dumps [7]. In this work we counteract the EMAN4 attack by our control flow policy. This policy only allows to fetch bytecodes which are inside the bytecode area.

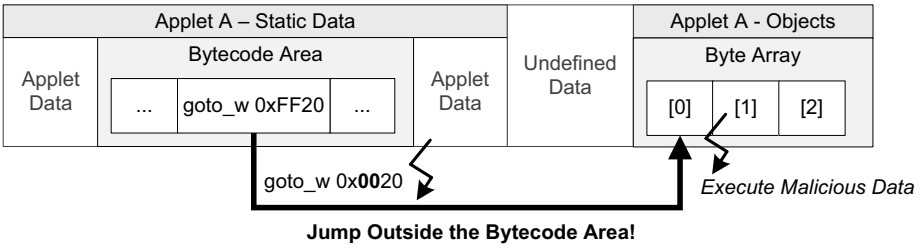

Fig. 1. The EMAN4 run-time attack changes the jump address 0xFF20 to 0x0020, which leads to the security threat of executing bytecode outside the defined BA of the current applet [6]

3 Defensive VM Layer

In this work, we propose adding a novel security layer to the Java Card. Through this layer, access to internal structures (e.g., OS, LV and BA) of the VM is handled. In reference to its defensive nature and its primary use for enabling a defensive VM, we name this layer the defensive VM (D-VM) layer. An overview of the D-VM layer and the D-VM API, which is used by the VM, is depicted in Figure 2 and is explained in detail below.

Functionalities offered by the D-VM API include, for example, pushing and popping data onto the OS, writing and reading from the LV and fetching Java bytecodes. It is possible for the VM to implement all Java bytecodes by using these API functions. The pseudo-code example in Listing 1.1 shows the process of fetching a bytecode and the implementation of the *sadd* bytecode using our D-VM API approach. The *sadd* bytecode pops two values of *integral data* type from the OS and pops the sum as an *integral data* type back onto the OS.

Listing 1.1. Pseudo-code of the VM using the API functions of the newly introduced D-VM layer.

```
 //use the D-VM API to fetch the next bytecode from the BA
switch(dvm_fetch_bytecode())
{
  ...
  case sadd: //implementation of the sadd bytecode.
  {
     //use the D-VM API to obtain the two values from the OS
    result = dvm_pop_integralData() + dvm_pop_integralData();
     //use the D-VM API to write the sum back onto the OS
    dvm_push_integralData(result);
  }
  ...
}
```

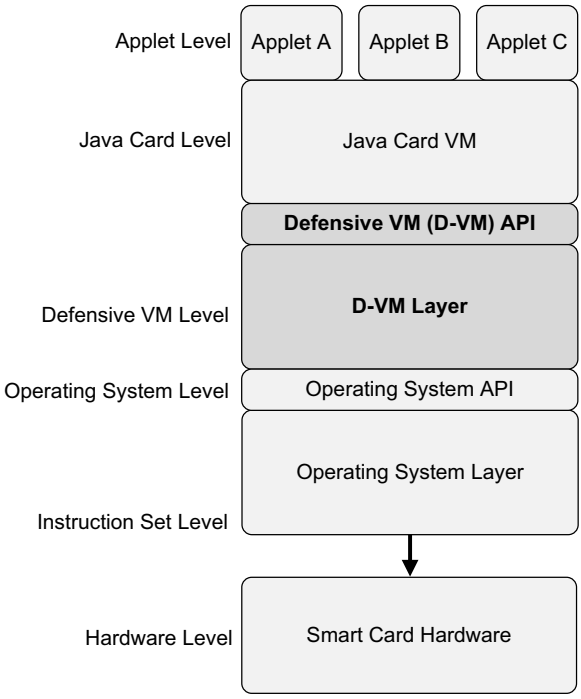

Fig. 2. The VM executes Java Card applets and uses the newly introduced D-VM layer to secure the Java Card against FAs

The security mechanisms within the security layer intended to protect the VM from FAs are hidden from the VM programmer. A security architect, specialized for VM security, is able to implement and choose the appropriate countermeasures within the D-VM layer. These countermeasures are based on state-of-the-art knowledge and the hardware constraints of the smart card architecture. Programmers implementing the VM do not need to know these security techniques in detail but rather just use the D-VM API functions.

If HW features are used, the D-VM layer communicates with these units and configures them through specific instructions. Through this approach, it is also very easy to alter the SW implementations by changing the D-VM layer implementation without changing specific Java bytecode implementations. It is possible to fulfill the same security policy on different smart card platforms where specific HW features are available.

On a code size-constrained smart card platform, an implementation that has a small code size but requires more main memory or execution time is used. The appropriate implementations of security features within the D-VM API are used without the need to change the entire VM.

Dynamic Countermeasures: The D-VM layer is also a further step to enable dynamic fault attack countermeasures such as that proposed by Barbu in [3]. In this work, he proposes a VM that uses different bytecode implementations depending on the actual security level of the smart card. If an attack or malicious behavior is detected, the security level is decreased. This decreased security leads to an exchange of the implemented bytecodes with more secure versions. In these more secure bytecodes, different additional checks, such as double reads, are implemented, which leads to decreased run-time performance.

Our D-VM layer further advances this dynamic countermeasure concept. Depending on the actual security level, an appropriate D-VM layer implementation is used. Therefore, the entire bytecode implementation remains unchanged, but it is possible to dynamically add and change security checks during run-time. An overview of this dynamic approach is outlined in Figure 3.

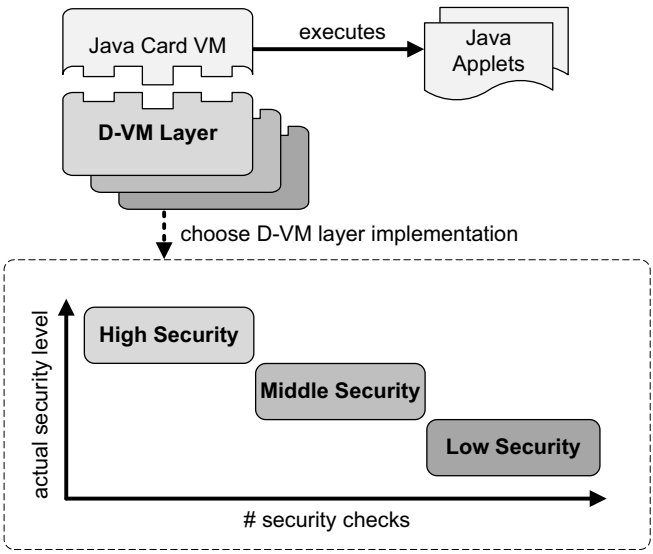

Fig. 3. Based on the current security level of the VM, an appropriate D-VM layer implementation is chosen

The actual security level of the card is determined by HW sensors (e.g., brightness and supply voltage) and the behavior of the executing applet. For example, at a high security level, the D-VM layer can perform a read operation after pushing a value into the OS memory to detect an FA. At a lower security level, the D-VM layer performs additional bound, type and control-flow checks.

Security Context of an Applet: Another use case for the D-VM layer is the post-issuance installation of applets on the card. We focus on the user-centric ownership model (UCOM) [1] in which Java Card users are able to load their own

applets onto the card. For the UCOM approach, each newly installed applet is assigned a defined security level at installation time. The security level depends on how trustworthy the applet is. For example, the security level for an applet signed with a valid key from the service provider is quite high, which results in a high execution speed. Such an applet should be contrasted with an applet that has no valid signature and is loaded onto the card by the Java Card owner. This applet will run at a low security level with many run-time checks but a slower execution speed. Furthermore, access to internal resources and applets installed on the card could be restricted by the low security level.

3.1 Security Policy

This chapter introduces the three security policies used in this work. With the help of these policies, it is possible to counteract the most dangerous threats that jeopardize security-critical data on the card. The type and bound policies are taken from [14] and are augmented with a control-flow policy. The fulfillment of the three policies on every bytecode is checked by three different D-VM layer implementations using our D-VM API.

Control-Flow Policy: The VM is only allowed to fetch bytecodes that are within the borders of the currently active method's BA. Fetching of bytecodes that are outside of this area is not allowed. The actual valid method BA changes when a new method is invoked or a return statement is executed. Because of this policy, it is no longer possible for control-flow changing bytecodes (e.g., *goto_w* and *if_scmp_w*) to jump outside of the reserved bytecode memory area. This policy counters the EMAN4 attack [6] on the Java Card and all other attacks that rely on the execution of a data array or code of an-other applet that is not inside the current BA.

Type Policy: Java bytecodes are strongly typed in the VM specification [19]. This typing means that for every Java bytecode, the type of operand that the bytecode expects and the type of the result stored in the OS or LV are clearly defined. An example is the *sastore* bytecode, which stores a *short* value in an element of a *short* array object. The *sastore* bytecode uses the top three elements from the OS as operands. The first element is the address of the array object, which is of type *reference*. The second element is the index operand of the array, which must be of type *short*. The third element is the value, which is stored within the array element and is of type *short*.

Type confusion between values of integral data (*boolean*, *byte* or *short*) and object references (*byte[]*, *short[]* or *class A*, for example) is a serious problem for Java Cards [24,17,13,25,6,11]. To counter these attacks, we divide all data types into the two main types, *integralData* and *reference*. Note that this policy does not prevent type confusion inside the main type *reference* between array and class types.

Bound Policy: Most Java Card bytecodes push and pop data onto the OS or read and write data into the LV, which can be considered similar to registers. The OS is the main component for most Java bytecode operations. Similar to buffer overflow attacks in C programs [9], it is possible to overflow the reserved memory space for the OS and LV. An adversary is then able to set the return address of a method to any value. Such an attack was first found in 2011 by Bouffard [6,7]. An overflow of the OS happens by pushing or popping too many values onto the OS. An LV overflow happens when an incorrect LV index is accessed. This index parameter is decoded as an operand for several LV-related bytecodes (e.g., *sstore*, *sload* and *sinc*). This operand is therefore stored permanently in the non-volatile memory. Thus, changing this operand through an FA gives an attacker access to memory regions outside the reserved LV memory region. These memory regions are created for every method invoked and are not changed during the method execution. Therefore in this work, we permit Java bytecodes to operate only within the reserved OS and LV memory regions.

4 Java Card Prototype Implementation

In this work three implementations of the D-VM layer are proposed to perform run-time security checks on the currently executing bytecode. Two implementations perform all checks in SW to ensure our security policies. One implementation uses dedicated HW protection units to accelerate the run-time verification process. The implementations of the D-VM layer were added into a Java Card VM and executed on a smart card prototype. This prototype is a cycle-accurate SystemC [12] model of an 8051 instruction set-compatible processor. All software components, such as the D-VM layer and the VM, are written in C and 8051 assembly language.

4.1 D-VM Layer Implementations

This section presents the implementation details for the three implemented D-VM layers used to create a defensive VM. All three implemented D-VM layers fulfill our security policy presented in Chapter 3 but differ from each other in the detailed manner in which the policies are satisfied. The key characteristic of the two SW D-VM implementations is that they use a different implementation of the type-storing approach to counteract type confusion. The run-time type information (*integralData* or *reference*) used to perform run-time checks can be stored either in a type bit-map (memory optimization) or in the actual word size of the microprocessor (speed optimization). The HW Accelerated D-VM uses a third approach and stores the type information in an additional bit of the main memory. Through this approach, the HW can easily store and check the type information for every OS and LV entry. An overview of how the type-storing policy is ensured by our D-VM implementations and a memory layout overview are shown in Figure 4 and explained in detail in the next sections.

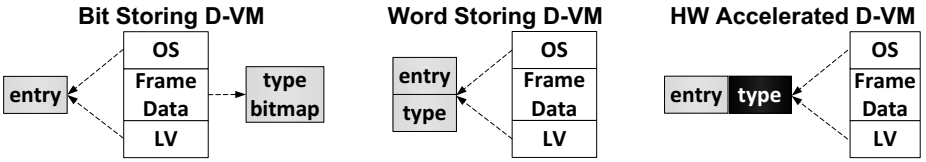

Fig. 4. The Bit Storing D-VM stores the type information for every OS and LV entry in a type bitmap. The Word Storing D-VM stores the type information below the value in the reserved OS and LV spaces. The HW Accelerated D-VM holds the type information as an additional type bit, which increases the memory size of a word from 8 bits to 9 bits.

Bit Storing D-VM: This D-VM layer implementation stores the type information for every element on the OS and LV in a type bitmap. The type information for every entry of the OS and LV is now represented by a one-bit entry. A problem with this approach is that the run-time overhead is quite high because different shift and modulo operations must be performed to store and read the type information from the type bitmap. These operations (shift and modulo) are, for the 8051 architecture, computationally expensive operations and thus lead to longer execution times. An advantage of the bit-storing approach is the low memory overhead required to hold the type information in the type bitmap.

Word Storing D-VM: The run-time performance of the type storing and reading process is increased by storing the type information using the natural word size of the processor and data bus on which the memory for the OS and LV is located. Every element in the OS and LV is extended with a type element of a word size such that it can be processed very quickly by the architecture. By choosing this implementation, the memory consumption of the type-storing process increases compared with the previously introduced SW Bit Storing D-VM. Pseudo-codes for writing to the top of the stack of the OS for the bit- and word-storing approach are shown in Listings 1.2 and 1.3.

Listing 1.2. Operations needed to push an element on the OS by the Bit Storing D-VM

```
dvm_push_integralData(value)
{
  //push value onto OS and
  //increase OS size
 OS[size++] = value;
  //store type information
  //into type bitmap,
  //INT->integralData type
 bitmap[size/8] = INT<<(size%8);
}
```

Listing 1.3. Operations needed to push an element on the OS by the Word Storing D-VM

```
dvm_push_integralData(value)
{
  //push value onto OS
  //increase OS size
 OS[size++] = value;
  //store type information
  //into next memory word,
  //INT->integralData type
 OS[size++] = INT;
}
```

HW Accelerated D-VM: Performing type and bound checks in SW to fulfill our security policy consumes a lot of computational power. Types must be loaded, checked and stored for almost every bytecode. The bounds of the OS and LV must be checked such that no bytecode performs an overflow. The HW Accelerated D-VM layer uses specific HW protection units of the smart card to accelerate these security checks. New protection units (bound protection and type protection) are able to check if the current memory move (MOV) operation is operating in the correct memory bounds. The type information for the OS and LV entries is stored as an additional type bit for every main memory word. The information is decoded into new assembly instructions to specify which memory region (OS, LV or BA) and with which data type (*integralData* or *reference*) the MOV operation should write or read data. An overview of the HW Accelerated D-VM is shown in Figure 5. Depending on the assembly instruction, the HW protection units perform four security operations:

- Check if the Java opcode is fetched from the current active BA.
- Check if the destination address of the operation is within the memory area of the OS or LV. If the operation is not within these two bounded areas, a HW security exception is thrown.
- For every write operation write the type decoded in the CPU instruction into the accessed memory word.
- For every read operation, check if the stored type is equal to the type decoded in the CPU instruction. If they are not equal, throw a hardware security exception.

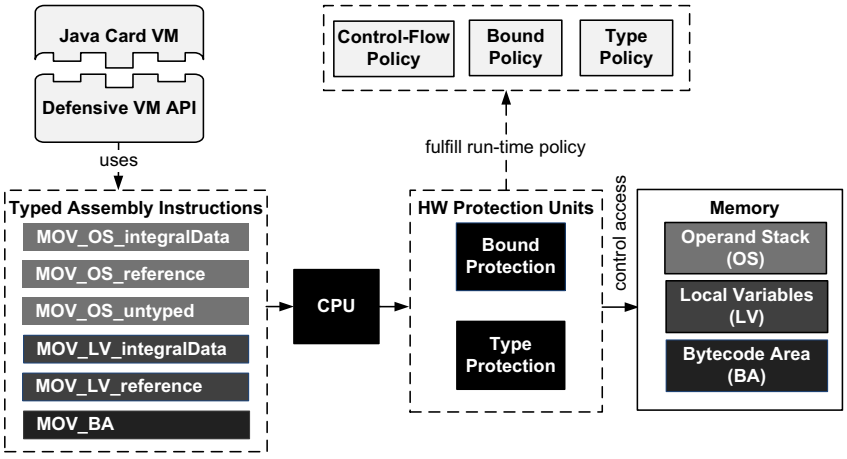

Fig. 5. Overview of the HW Accelerated D-VM implementation using new typed assembly instructions to access VM resources (OS, LV and BA). Malicious Java bytecodes violating our run-time policy will be detected by new introduced HW protection units.

5 Prototype Results

In this section, we present the overall computational overhead of the three implemented D-VM layers and their main memory consumption. All of them are compared with a VM implementation without the D-VM layer. The speed comparison is performed for different groups of bytecodes by self written micro-benchmarks where all bytecodes under test are measured. These test programs first perform an initialization phase where the needed operands for the bytecode under test are written into the OS or LV. After the execution of the bytecode under test the effects on the OS or LV are removed. Note that our smart card platform has no data or instruction cache. Therefore, no caching effects must be taken into account for all test programs.

5.1 Computational Overhead

Speed comparisons for specific bytecodes are shown in Figure 6. For example, the Java bytecode *sload* requires 148% more execution time for the Word Storing D-VM. For the Bit Storing D-VM, the execution overhead is 212%. The increased overhead is because of the expensive calculations used to store the type information in a bitmap. For the HW Accelerated D-VM, the execution speed decreases by only 4% because all type and bound checks are performed using HW. Additional run-time statistics for groups of bytecodes are listed in Table 1. As expected, the Bit Storing D-VM consumes the most overall run-time, with an increase of 208%. The Word Storing D-VM needs 142% more run-time. The HW Accelerated D-VM has only 6% more overhead.

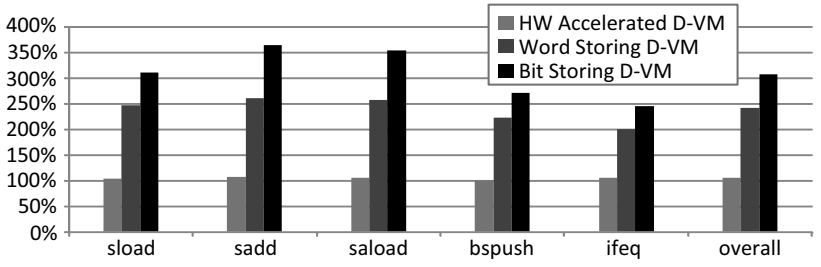

Fig. 6. Speed comparison of individual bytecodes for the different D-VM layer implementations proposed in this work. The results are compared with a VM without the D-VM layer.

5.2 Main Memory Consumption

The HW Accelerated D-VM requires one type bit per 8 bits of data to store the type information during run-time. This results in an overall main memory increase of 12.5%. The Word Storing D-VM requires in the worst case 33% more memoy because one type byte holds the type information for two data bytes.

Table 1. Speed comparison for different groups of bytecodes compared with a VM without the D-VM layer

Bytecode Groups	HW Accelerated D-VM	Word Storing D-VM	Bit Storing D-VM
Arithmetic/Logic	+7%	+146%	+240%
LV Access	+5%	+185%	+243%
OS Manipulation	+5%	+151%	+231%
Control Transfer	+7%	+113%	+173%
Array Access	+5%	+130%	+166%
Overall	+6%	+142%	+208%

The Bit Storing D-VM requires approximately 6.25% more memory in the case in which the entire memory is filled with OS and LV data. This is because the Bit Storing D-VM requires one type bit per 16 bits of data.

6 Conclusions and Future Work

This work presents a novel security layer for the virtual machine (VM) on Java Cards. Because it is intended to defend against fault attacks (FAs), it is called the defensive VM (D-VM) layer. This layer provides access to security-critical resources of the VM, such as the operand stack, local variables and the bytecode area. Inside this layer, security checks, such as type checking, bound checking and control-flow checks, are performed to protect the card against FAs. These FAs are executed during run-time to change the control and data flow of the currently executing bytecode.

By storing different implementations of the D-VM layer on the card, it is possible to choose the appropriate security implementation based on the actual security level of the card. Through this approach, the number of security checks can be increased during run-time by switching among different D-VM implementations. Furthermore, it is possible to assign a trustworthy applet a low security level, which results in high execution performance, and vice versa. One D-VM layer implementation can be, for example, low security with high execution speed or high security with low execution speed. Another advantage is the concentration of the security checks inside the layer.

To demonstrate this novel security concept, we implemented three D-VM layers on a smart card prototype. All three layers fulfill the same security policy (control-flow, type and bound) for bytecodes but differ in their implementation details. Two D-VM layer implementations are fully implemented in software but differ in the manner in which the type information is stored. The Bit Storing D-VM has the highest run-time overhead, 208%, but the lowest memory increase, 6.25%. The Word Storing D-VM decreases the run-time overhead to 142% but consumes approximately 33% more memory. The HW Accelerated D-VM uses dedicated Java Card HW to accelerate the run-time verification process and has an execution overhead of only 6% and a memory increase of 12.5%.

In future work, we will focus on the question of which sensor data should be used to increase the internal security of the Java Card. Another question is how many security states are required and how much they differ in their security needs.

Acknowledgments. The authors would like to thank the Austrian Federal Ministry for Transport, Innovation, and Technology, which funded the CoCoon project under the FIT-IT contract FFG 830601. We would also like to thank our project partner NXP Semiconductors Austria GmbH.

References

1. Akram, R., Markantonakis, K., Mayes, K.: A Paradigm Shift in Smart Card Ownership Model. In: 2010 International Conference on Computational Science and Its Applications (ICCSA), pp. 191–200 (March 2010)
2. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. Proceedings of the IEEE 94(2), 370–382 (2006)
3. Barbu, G., Andouard, P., Giraud, C.: Dynamic Fault Injection Countermeasure. In: Mangard, S. (ed.) CARDIS 2012. LNCS, vol. 7771, pp. 16–30. Springer, Heidelberg (2013)
4. 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)
5. 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)
6. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.-L.: Combined Software and Hardware Attacks on the Java Card Control Flow. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 283–296. Springer, Heidelberg (2011)
7. Bouffard, G., Lanet, J.-L.: The Next Smart Card Nightmare. In: Naccache, D. (ed.) Cryphtography and Security: From Theory to Applications. LNCS, vol. 6805, pp. 405–424. Springer, Heidelberg (2012)
8. 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)
9. Cowan, C., Wagle, P., Pu, C., Beattie, S., Walpole, J.: Buffer overflows: attacks and defenses for the vulnerability of the decade. In: Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems], pp. 227–237 (2003)
10. Dubreuil, J., Bouffard, G., Lanet, J.-L., Cartigny, J.: Type Classification against Fault Enabled Mutant in Java Based Smart Card. In: 2012 Seventh International Conference on Availability, Reliability and Security (ARES), pp. 551–556 (August 2012)
11. Hamadouche, S., Bouffard, G., Lanet, J.-L., Dorsemaine, B., Nouhant, B., Magloire, A., Reygnaud, A.: Subverting Byte Code Linker service to characterize Java Card API. In: Proceedings of the 7th Conference on Network and Information Systems Security (SAR-SSI), pp. 122–128 (2012)

12. IEEE: Open SystemC Language Reference Manual IEEE Std 1666-2005, IEEE
13. Iguchi-Cartigny, J., Lanet, J.-L.: Developing a Trojan applets in a smart card. Journal in Computer Virology 6, 343–351 (2010)
14. Lackner, M., Berlach, R., Loinig, J., Weiss, R., Steger, C.: Towards the Hardware Accelerated Defensive Virtual Machine – Type and Bound Protection. In: Mangard, S. (ed.) CARDIS 2012. LNCS, vol. 7771, pp. 1–15. Springer, Heidelberg (2013)
15. Leroy, X.: Bytecode verification on Java smart cards. Software: Practice and Experience 32(4), 319–340 (2002)
16. Markantonakis, K., Mayes, K., Tunstall, M., Sauveron, D., Piper, F.: Smart card security. In: Nedjah, N., Abraham, A., de Macedo Mourelle, L. (eds.) Computational Intelligence in Information Assurance and Security. SCI, vol. 57, pp. 201–233. Springer, Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-71078-3_8
17. 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)
18. Oracle: Runtime Environment Specification. Java Card Platform, Version 3.0.4, Classic Edition (2011)
19. Oracle: Virtual Machine Specification. Java Card Platform, Version 3.0.4, Classic Edition (2011)
20. Razafindralambo, T., Bouffard, G., Thampi, B.N., Lanet, J.-L.: A Dynamic Syntax Interpretation for Java Based Smart Card to Mitigate Logical Attacks. In: Thampi, S.M., Zomaya, A.Y., Strufe, T., Alcaraz Calero, J.M., Thomas, T. (eds.) SNDS 2012. CCIS, vol. 335, pp. 185–194. Springer, Heidelberg (2012)
21. Sauveron, D.: Multiapplication smart card: Towards an open smart card? Information Security Technical Report 14(2), 70–78 (2009); Smart Card Applications and Security
22. Séré, A.A.K., Iguchi-Cartigny, J., Lanet, J.-L.: Checking the Paths to Identify Mutant Application on Embedded Systems. In: Kim, T.-H., Lee, Y.-H., Kang, B.-H., Ślęzak, D. (eds.) FGIT 2010. LNCS, vol. 6485, pp. 459–468. Springer, Heidelberg (2010)
23. Séré, A.A.K., Iguchi-Cartigny, J., Lanet, J.-L.: Evaluation of Countermeasures Against Fault Attacks on Smart Cards. International Journal of Security and Its Applications 5(2), 49–61 (2011)
24. 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)
25. Vetillard, 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)