Towards the Hardware Accelerated Defensive Virtual Machine - Type and Bound Protection

Michael Lackner[1], Reinhard Berlach[1], Johannes Loinig[2], Reinhold Weiss[1], and Christian Steger[1]

[1] Institute for Technical Informatics,
Graz University of Technology, Graz, Austria
{michael.lackner,reinhard.berlach,rweiss,steger}@tugraz.at
[2] NXP Semiconductors Austria GmbH, Gratkorn, Austria
johannes.loinig@nxp.com

Abstract. Currently, security checks on Java Card applets are performed by a static verification process before executing an applet. A verified and later unmodified applet is not able to break the Java Card sand-box model. Unfortunately, this static verification process is not a countermeasure against physical run-time attacks corrupting the control or data flow of an applet. In this piece of work, designs for Java Card Virtual Machines are investigated in relation to their ability to perform run-time security checks. These security checks are accelerated by hardware units and performed in parallel to CPU instructions that are executing concurrently. Attacks on the Java operand stack and local variables, which are elementary components for the Virtual Machine, are thwarted by type and bound protection. To enable these hardware checks, different designs of a defensive Java Card Virtual Machine are compared to their overheads on a prototype platform.

1 Introduction

Current applied static verification of Java Card applets provides insufficient security protection against run-time Fault Attacks. This is especially a problem in the field of multi-application Java Cards. In this field, cards are used in a wide range of applications (e.g., passport, e-money) and have the ability to perform post-issuance loading of new applets. An adversary provoking a Logical Attack by changing the bytecode or internal representation of an uploaded Java applet can get access to security related data from other applets or the Java Card Virtual Machine (JCVM) [12]. To thwart Logical Attacks, verification of applets is performed either off-card or on-card. This verification procedure is currently a static process performed once before an applet is executed. One of the most time and memory consuming checks performed, is the bytecode verification process [9, 15] which is based on a data flow and control flow analysis.

Java Card applets are stored into non-volatile memories such as EEPROM. With the help of physical Fault Attacks it is possible for a JCVM to read out incorrect values from these memories or skip CPU instructions. Therefore, it

is possible to change the bytecode of stored applets to execute ill-formed Java instructions. Knowledge about these attack possibilities is used in [3] to create a new class of attacks called Combined Attacks. To perform Combined Attacks, applets which pass the verification process are used and become malicious in combination with a Fault Attack. Combined Attacks are used to bypass the Java Card sand-box, mounted by the static verification process, JCVM and Java Card Runtime Environment [11].

To guard the Java Card against run-time attacks, a so called defensive JCVM is needed [4]. This defensive JCVM can be reached by performing all checks done by the static verification process during run-time. However, this is currently not achievable because of the constrained hardware resources of today's Java Cards. In this work specific security checks, extracted from the Java Card specification [12], are performed on the executing bytecode during run-time. In this specification the data flow is exactly defined for every bytecode with some additional constraints which must be fulfilled.

To speed up bytecode checking during run-time, new hardware protection units are introduced in this work to speed up the checks performed on every bytecode. These hardware checks can be performed in parallel while the CPU performs its operations. Therefore hardware checks are a good solution for run-time checks that are performed very often, in contrast to software checks. Software checks slow down the whole system if they are performed on the same CPU that the standard operations are performed on. Beside this benefit, hardware checks also have the advantage of being more immune against additional fault injects onto the Java Card. This is due to the fact that software checks [13, 5, 2] are vulnerable to skipping them by additional Fault Attacks. This threat of skipping software security operations leads Vertanen in [16] to the conclusion that hardware assisted run-time checks are mandatory for enhancing run-time security for Java Cards.

This work introduces a hardware accelerated defensive JCVM which performs selected security checks on the executing bytecodes by hardware with a low computational overhead. These checks are also part of the verification process which is done statically before executing a Java applet. As far as we know, such a hardware accelerated defensive JCVM has not been introduced in literature before.

The contribution of this work is the definition of a run-time security policy extracted from the Java Card specification [12] to ensure that the executing bytecode performs valid operations. This run-time policy prevents type confusion and overflow/underflow attacks on the operand stack (OS) and local variable (LV) memory inside the JCVM. With this policy it is not possible for an adversary to perform type confusion between values of type *integralData* and object *references* on the OS and LV. Furthermore, two hardware accelerated defensive JCVM designs are presented with their main parts, such as additional hardware protection units and new CPU instructions. The new CPU instructions are used inside the JCVM to process bytecodes and communicate directly with the

hardware protection units leading to a very low computational overhead. This communication is depicted in Figure 1.

Section 2 gives an overview of attacks on Java Cards, bytecodes violating the Java frame bounds and how to enable a defensive JCVM. Section 3 describes the security policy and the design of all defensive JCVMs introduced in this work. Section 4 presents the prototype implementation of these designs on a SystemC 8051 derivate. Section 5 analyses the run-time costs on execution speed and hardware changes needed to activate our JCVM designs. Finally, conclusions and future work are drawn in Section 6.

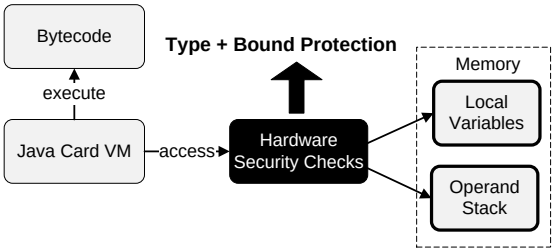

Fig. 1. In this work the operand stack and the local variables are protected during run-time by hardware accelerated security checks.

2 Related Work

In this section an overview of possible attacks on the Java Card is given with focus on run-time fault attacks. Following this, previous work on run-time countermeasures and an overview of defensive JCVMs are presented.

2.1 Attack Overview

Attack scenarios on Java Cards are manifold [18, 19]. Side Channel Attacks are used to draw conclusions of internal operations by studying physical phenomena of the chip. Invasive Attacks are used for optical or measurement analysis of internal components. Fault Attacks (FA) change the physical environment of the chip under attack [1]. These are for example, temperature changes, additional light of a laser or spikes in the power supply or clock source. These FA lead to an undefined behavior of the chip by skipping instructions or read/write errors to memory like the EEPROM. This is especially a problem for post-issuance loaded applets due to the fact that they are mostly installed in non-volatile memory. Therefore, a FA during the fetch process of a JCVM can lead to ill-formed applets even if a static verification was performed. This ill-formed code enables an adversary to circumvent the Java Card security model and enables an applet to have access to unauthorized resources. This security problem of FA to

verified applets is well known in literature and is used to enable different attack paths [10, 3, 17, 14].

2.2 Frame Bound Violation Attacks

Generally, inside every Java Frame, specific memory areas are reserved for OS, LV and internal frame data. Every time a new Java method is invoked, the JCVM creates a new frame and pushes it onto the Java stack. Specific implementation details for the Java Frame are not provided by Java Card specification. Therefore, the specific frame data depends on the particular implementation. In general the frame data contains a return address so that it is possible to return to the code of the old frame. The size needed for a frame and all its containing elements (e.g., OS, LV) is ascertained when the method is invoked and is not changing during method execution.

Ill formed bytecode can now access illegal memory regions by performing an OS or LV out of bound access as illustrated in Figure 2. In [5] an attack called EMAN2 was performed. There an invalid LV index was used by an ill formed bytecode *sstore* to access the memory region of the frame data where the return address of the current frame is stored. With the help of this ill formed bytecode, an adversary can set the return address to any value. In their attack the return address was set to the address of an array which leads to the security threat of executing adversary definable data. This illegal execution of data opens new security issues not treated here in detail. The threats of OS overflow/underflow and bytecodes using invalid LV index are thwarted by the run-time policy of this work.

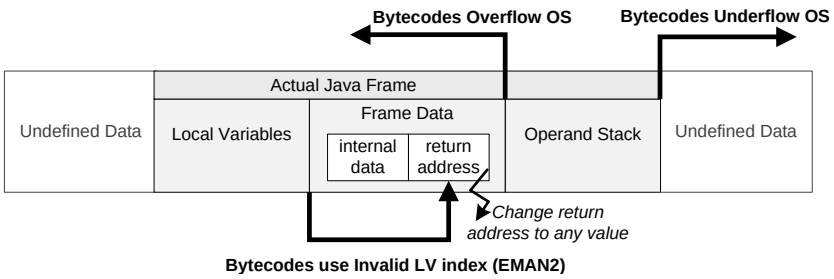

Fig. 2. OS overflow and underflow leads to illegal memory access outside the reserved OS memory space. An adversary who uses bytecode with invalid LV index can overwrite the return address of the current active frame [5].

2.3 Enabling a Defensive Virtual Machine

Currently the Java Card research community concentrates on finding attack paths to bypass the Java Card security model by FAs and Combined Attacks.

[3, 10, 17, 14]. Also a lot of effort is invested in exploiting and thwarting Side Channel Attacks [8]. In contrast to these big research topics, the question of how to enable a defensive JCVM is a research topic with little public attention. However, in the Java Card industry the know-how to enable defensive JCVM designs is of course available. This fact is proven by different works bringing the defensive nature of current available industrial Java Card products to light [10, 7]. Techniques and knowledge that provide such a defensive design are of course not freely available. Currently research related to FA countermeasures is focussed on static verification of an applet and checking that the exact verified code is executed. This can be done by code integrity and control flow checks in software (SW) during run-time [13, 5]. The annotations that enable these checks are stored in an additional component of a verified CAP-file. Research was also done to check the OS integrity against FA by performing double reads by SW [2].

This work focuses on a hardware accelerated defensive JCVM performing security checks during run-time. Based on a policy it checks if the executing bytecode is behaving correctly. Compared to current countermeasures in literature the approach in this work does not just check the integrity of the bytecode or OS, it performs checks based on a policy. This approach stops either manipulated applets loaded onto the card or run-time FA from violating this policy. Furthermore, performing these checks in hardware makes it more resistant to additional FA which are also able to skip additional software checks.

3 Design of the Defensive Virtual Machine

In this section the run-time security policies for all defensive JCVM designs in this work are shown. This is followed by our method of reducing all Java data types to two main types.

3.1 Defensive Run-time Policy

The OS and LV, located in the Java Frame, are main parts of the JCVM. The JCVM is a stack machine and performs most operations on the OS. Therefore, securing these parts of the JCVM are the first steps to hampering or stopping Fault Attacks during run-time. The following two main policies for the OS and LV are retained by our defensive designs during run-time:

- **Frame Type Policy:** All bytecodes which access the OS or LV must use the right main data type (*integralData* or *reference*) which is expected by the bytecode during its execution. In this work all numerical types are combined (*boolean*, *byte*, *short*) to the main data type *integralData*. All object references (e.g., *short array*, *byte array*, *Class A*) are combined to the main data type *reference*.
- **Frame Bound Policy:** Bytecodes operating on the OS or LV are not allowed to access data outside the frame bounds. This means that bytecodes are not allowed to overflow or underflow the OS. Furthermore, all bytecodes accessing the LV must be inside the borders of the reserved LV memory area.

Policy Creation: The two policies above were extracted from the JCVM specification [12] where for every bytecode a textual description of the operation is given. In this specification it is for every bytecode defined from which JCVM component needed operands are taken and results are written back. Also the type information is specified for every operand and result value. Such a bytecode specification for the *sstore* instruction is listed below. This bytecode consists of two bytes, the opcode (0x29) and an index referencing to an item of the LV.

> *"The index is an unsigned byte that must be a valid index into the local variables of the current frame (Section 3.5, "Frames"). The value on top of the operand stack must be of type short. It is popped from the operand stack, and the value of the local variable at index is set to value."* [12]

The requirement that bytecodes perform no OS stack overflow/underflow, access the right LV index and operate with the right types on the OS and LV is crucial for the security concept of the Java Card and therefore checked by all defensive JCVM designs of this work.

3.2 Design of the Defensive JCVMs

In this section we introduce two designs for a defensive JCVM which fulfill the security policies defined in the previous section. A general overview of the defensive JCVM designs is shown in Figure 3 and described how they are used in more detail below. Note that our defensive JCVM designs are not able to thwart all sort of attacks on a Java Card. Examples of such undetected attacks are control flow changes (skipping a branch instruction) or data corruption (read corrupted values from the RAM).

- **Type Storing:** Every entry on the OS or LV is extended with type information in order to distinguish between *integralData* and *reference* during run-time. During run-time it is now possible to check if the expected type for the bytecode is on the OS or LV which is the obvious defensive approach to enable a Defensive JCVM. A disadvantage of this approach is the additional memory needed for type storing and the computational overhead to perform type checking.
- **Type Separating:** Every Java main data type (*integralData* and *reference*) operates on its own OS and LV memory area. No general OS and LV area where all data types occur exists. Type confusion between the two main data types is therefore no longer possible during run-time because every bytecode always receives the right type.
  A disadvantage of the Type Separating design is that an attack is only detected by its security related side effects on the current frame. Such a side effect is for example an OS underflow for a specific type.

3.3 Two Types for Type Storing and Type Separating

In this section we introduce our approach for separating the Java Card types into two main data types to enable the Type Storing and Type Separating JCVM

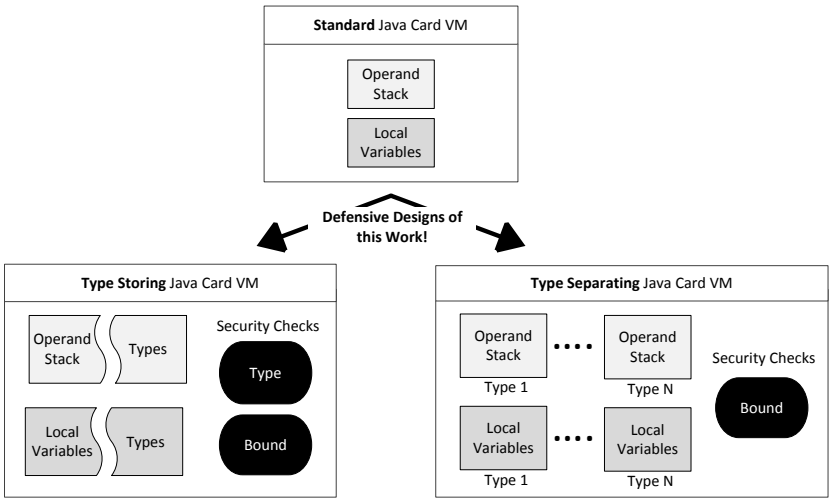

Fig. 3. In this work two designs are used to fulfill the run-time security policy.

that was presented in the previous chapter. Java bytecodes are highly typed. This means that based on the data type different opcodes exist for the same operation [12, Table 3-1]. For example, only the *sstore* bytecode is allowed to push integral data types (boolean, byte, short) into the LV. Another Java bytecode is used to store an element of type *reference*, pointing to an object, into the LV. It is therefore possible to differentiate between two main data types and distinguish them just by looking at the bytecode. In this work they are called *integralData* and *reference*:

- ***integralData*** These are the primitive constant data types that represent the numerical values of the JCVM: *boolean*, *byte*, *short*. Elements of this data type can be deliberately created by executing the bytecode *sconst_1*. This bytecode pushes an integral value 1 with type *short* on the OS.
- ***reference*** These are all kinds of *references* to objects and the *returnAddress* type to enable sub-routines. An applet programmer can only indirectly create elements of type *reference*. For example the bytecode *new_array* pushes the reference of a newly created array object onto the OS. This address can have any logical structure and does not have to correlate with physical addresses of objects stored on the card. For example the JCVM can create a random number and a look up table maps this number to a real memory address.

By separating these two main data types it is no longer possible for an adversary to create object references with a defined value by confusing *integralData* and *reference*. It is also not possible for an adversary to get deeper insight into how the JCVM represents references. For this insight an adversary would have to perform type confusion between *reference* and *integralData* by sending the reference out of the card from the APDU buffer. The APDU class is responsible for receiving and sending data to off-card applications.

Thwarted Threats in Literature This sort of type confusion between integral data and object references is well known and often used as the first step of an attack path [16, 10, 7, 5, 17]. This attack path can enable an adversary creating self mutable code by executing data from a Java array [7, 5] or even gain access to forbidden methods of objects [17].

Note that type confusion between different objects such as *short array* and *Class A* object is not detected by the defensive JCVM designs in this work. This is because all object references are assigned to the *reference* main data type and cannot be distinguished. This determination also applies to the main data type *integralData* where it is not possible to detect type confusion between *byte* and *short*.

4 Prototype Implementation

In this work five different prototype JCVMs were implemented in C and assembly language. The JCVMs are based on the Classic Edition of the Java Card specification [12]. The hardware (HW) platform on which they run is an 8-bit Smart Card model written in SystemC [6]. This model is memory and instruction cycle accurate. Into this HW platform new typed CPU instructions were implemented. Furthermore, additional HW protection units were added to enable HW accelerated security checks for bytecodes accessing the OS and LV memory area.

4.1 Additional CPU Instructions

The information decoded into the new CPU instructions is illustrated in Figure 4. New typed CPU instructions are used by our JCVMs to process the bytecodes and perform access to the OS and LV memory regions. These decoded pieces of information are the access type (Read, Write), the destination of the accessing memory (OS, LV) and the type which should be written/read (*integralData*, *reference*, *untyped*). With the help of these pieces of information the protection units are able to check if the new CPU instruction doesn't perform a security policy violation during run-time. Such a violation is for example a LV element address which is outside the actual LV memory bounds.

Two examples of how to use these new instructions inside the JCVM program code in order to process the Java bytecodes is outlined in Figure 5. The *sadd* bytecode first reads two values from the OS by the new CPU instruction *Read_OS_integralData*. The result of the addition of these values is then written back by the instruction *Write_OS_integralData*. Another example is the bytecode *astore_0*. This bytecode uses the CPU instruction *Read_OS_reference* to read a reference value from the OS and stores it into the LV by the instruction *Write_LV_reference*. The big advantage in the sense of computational overhead of the new typed CPU instructions is that the JCVM can communicate very effectively with the HW protection units by using the new CPU instructions.

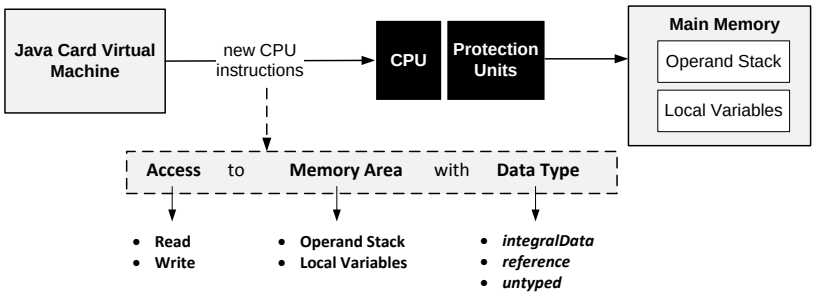

Fig. 4. The prototype JCVMs proposed in this work uses new assembly instructions to access the run-time protected OS and LV memory regions.

```
sadd:                              //Add two short values on the OS
   short VAR1, VAR2, SUM;          //Create variables for sadd
   VAR1 = Read_OS_integralData;    //Read first operand from OS
   VAR2 = Read_OS_integralData;    //Read second operand from OS
   SUM  = VAR1 + VAR2;             //Sum the two operands
   Write_OS_integralData = SUM;    //Write the sum back onto the OS

astore_0:                          //Store reference from OS to LV
   short REF1;                     //Create variables for astore_0
   REF1 = Read_OS_reference;       //Read reference from OS
   Write_LV_reference(0) = REF1; //Write reference into LV element 0
```

Fig. 5. Pseudocode example of the two bytecodes *sadd* and *astore*, processed by the JCVM. The JCVM uses our new CPU instructions to access the OS and LV memory.

4.2 Additional Hardware Protection Units

An overview of the new CPU instructions and the protection units needed to activate the Type Storing and Type Separating JCVM is presented in Figure 6. Based on the new CPU instructions introduced in the previous section, our HW protection units restrict the access to the security critical memory regions of the OS and LV. The Type Storing JCVM needs a type protection unit to check if the type expected by the bytecode is also available on the OS or LV.

- **Bound Protection Unit (BPU):** This unit is responsible for thwarting attacks performing an OS overflow or underflow. Furthermore, all bytecodes accessing the LV using a wrong index are detected. The BPU is used by the Type Storing and Type Separating JCVM prototype.
- **Type Protection Unit (TPU):** The TPU is responsible for checking that the bytecodes that are accessing the OS and LV are operating with the right data type. The TPU is only needed to enable the Type Storing JCVM.

4.3 Type Storing JCVM Implementation Details

To enable a Type Storing JCVM, the TPU must store additional type information for every element held by the OS and LV. Due to the fact that these two

parts are located in RAM, one additional type bit was added to every 8-bit word. This bit enables the distinction between the two main data types *integralData* and *reference*.

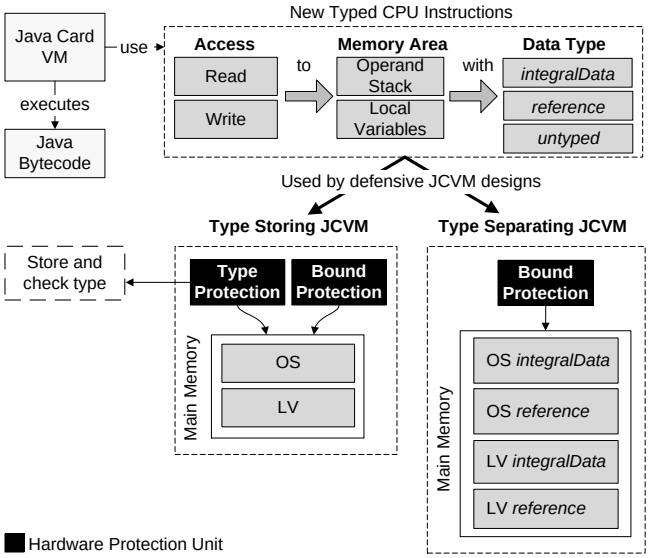

Fig. 6. Implementation overview of the two JCVM prototypes. Hardware protection units perform run-time checks on the OS and LV.

4.4 Type Separating JCVM Implementation Details

In this section we give insight into the detail of how the Type Separating JCVM was implemented and describe a tool chain to enable it. The Type Separating JCVM performs all bytecode operations on the right typed OS and LV. This Type Separating approach avoids type confusion. The type checking problem is reduced to a bound checking problem.

Most bytecodes work well with our run-time type separating approach to two main data types (*integalData*, *reference*). An exception are the bytecodes operating with undefined types on the OS: *pop*, *pop2*, *dup*, *dup2*, *dup_x* and *swap_x*. In this paper we call them untyped bytecodes. For these untyped bytecodes the JCVM does not know on which of the two separated OS, specific operations are performed during run-time.

As a solution for this problem the missing type information was added directly into the bytecode by using unused bytecodes which are not defined in the Java Card specification. For example, the *pop* instruction is either convert to *pop_reference* or *pop_integralData*. The JCVM now knows right after fetching a new bytecode, which typed OS it must operate on. Therefore, no execution

speed is wasted searching for the type information in additional components uploaded on the card. In the JCVM specification [12] only 185 (0x00 to 0xb8) of all available 8-bit bytecodes are specified. The unused bytecodes from 0xb9 to 0xfd can be used to decode the operand stack type information that is needed directly into the instruction.

To perform the exchange of untyped bytecodes with new typed bytecodes we propose a static replacement process performed once for every method. To speed up this process, type information obtained during the bytecode verification process can be used to exchange the untyped bytecodes with typed ones. However, in this work the replacement process for the untyped bytecodes is not looked at in detail.

5 Prototype Results and Discussion

In this section we show the computational overhead coming from full software (SW) implementations compared to running our HW accelerated prototypes. The SW implementations perform the same security checks that are performed by the HW protection units. Furthermore, the additional hardware overhead is compared between all prototypes.

5.1 Computational Overhead

Performing all security checks in SW increases the computational overhead significantly for frequently executed bytecodes, as illustrated in Figure 7. For example the *sload* bytecode executed by a Type Storing prototype in SW has a computational overhead of around 115% caused by the following run-time SW operations:

- Check if the index parameter to the LV is valid.
- Check if the element at the LV index is of type *integralData*.
- Check if pushing a value from the LV index to the OS provokes an overflow.
- Store the fact that the new value on the OS is of type *integralData*.

If the *sload* bytecode is executed on a HW accelerated prototype the overhead decreases to 5%. In Table 1 different groups of bytecodes are compared to their computational overhead. As expected, the HW accelerated prototypes consume much less computational overhead compared to prototypes which implement the checks in SW.

5.2 Hardware Overhead

In this section we give an overview of the HW modifications used to activate our HW accelerated prototypes, as depicted in Table 2. The instruction set of a standard 8051 microcontroller consists of 255 opcodes. Adding our new CPU instructions means an overall CPU instruction increase of around only 3,5%. Another important hardware modification is that the RAM module of the HW

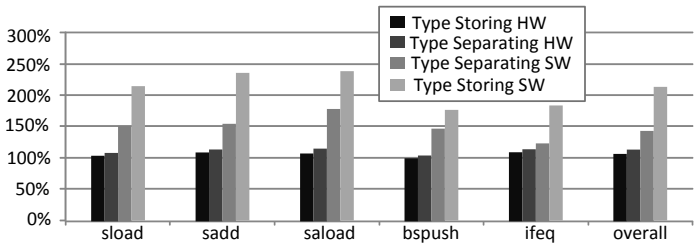

Fig. 7. Run-time measurement for specific bytecodes and the overall time of all implemented bytecodes for different JCVM implementations. Measurements are normalized to a JCVM without any run-time security checks.

Table 1. Computational overhead for all prototypes, normalized to a JCVM without performing run-time security checks.

	Type Storing		Type Separating	
Bytecode Groups	HW	SW	HW	SW
1: Arithmetic/Logic	+7%	+123%	+7%	+47%
2: Local Variable Access	+5%	+152%	+9%	+52%
3: Operand Stack Manipulation	+5%	+119%	+3%	+54%
4: Control Transfer	+8%	+77%	+9%	+23%
5: Array Creation/Manipulation	+6%	+111%	+9%	+59%
Overall	+6%	+115%	+8%	+42%

accelerated Type Storing JCVM was extended with an additional type bit for every memory word in order to differ between the main data types *integralData* und *reference*. Therefore, overall RAM memory size increases to 12,5%.

Table 2. HW modifications needed to activate the HW accelerated run-time security checks of the prototypes.

Additional Hardware	Type Storing	Type Separating
New 8051 CPU Instructions	9 (+3,5%)	8 (+3,1%)
New 8-bit Control Registers (SFRs)	11 (+52,4%)	15 (+71,4%)
New Bound Protection Unit (BPU)	Yes	Yes
New Type Protection Unit (TPU)	Yes	No
Extend RAM word with type bit	Yes	No

5.3 Type Confusion Attack Example

An example of a run-time attack on the Java Card prototypes in order to perform type confusion between *integralData* and *reference* is illustrated in Figure 8. There, a run-time attack changes the *bspush* code 0x10 0x19 to 0x00 0x19. The JCVM interprets 0x00 as NOP instruction and performs no action. The following

byte 0x19 is interpreted as the bytecode *aload_1* which pushes an array reference onto the OS. The *sreturn* instruction would now take the array reference and push it back to the calling function. An adversary is now able to use the array *reference* as an *integralData* which enables different attack paths such as executing the data inside the array [16, 10, 7, 5, 17]. Both defensive JCVMs designs from this work are able to thwart this type confusion attack on the OS between *integralData* and *reference*.

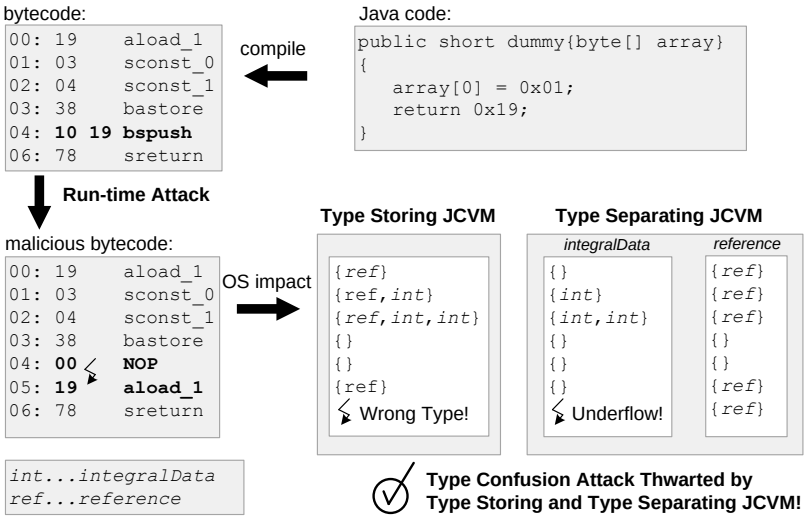

Fig. 8. Run-time type confusion attack on the OS to receive the address of an array. All defensive JCVM implementations of this work are able to thwart this attack.

Type Storing: By using the new typed CPU instructions, it is decoded inside the JCVM code that the *sreturn* instruction expects a value of type *integralData* on the OS. The previously executed instruction *aload_1* pushed the reference of an array with type *reference* on the OS. Therefore, the TPU hardware module finds the wrong type for the *sreturn* bytecode on the OS and throws a security exception.

Type Separating: Both main data types have their own OS and LV memory areas during run-time. Therefore, the malicious instruction *aload_1* pushes an array reference onto the *reference* OS containing all values from type *reference*. The *sreturn* bytecode tries to pop data from the *integralData* OS which is empty. The return operation is now aborted by a security exception because an underflow on the *integralData* OS is detected by the BPU hardware module. All type confusion attacks are avoided by the type separating architecture of the JCVM.

6 Conclusion and Future Work

This work presents Java Card Virtual Machine (JCVM) designs to counter different Fault Attacks. This is done by performing run-time security checks based on a security policy for each bytecode. These policies ensure that each bytecode which operates on the operand stack or the local variables memory area uses the right data type (*integralData* or *reference*). Furthermore bytecodes which overflow or underflow the OS or LV are detected. These run-time checks are accelerated by hardware protection units to make it harder to skip these checks with additional Fault Attacks. Furthermore, the defensive JCVM designs are profiting from the parallel execution of the hardware checks by having very low computational overhead.

In this work the design of a Type Storing and Type Separating JCVM were shown. Both designs were implemented on a Java Card prototype platform with several additional hardware changes. The requirements of the hardware to enable run-time checking by hardware protection units were listed for both defensive JCVM designs. We measured that these hardware accelerated prototypes consume 6% and 8% more execution time overall compared to a JCVM without any additional run-time security checks. This overhead is very low compared to prototypes which perform all run-time security checks in software and consume around 115% and 42% more execution time. Therefore, we have shown that our approach of performing additional security checks during run-time by using hardware units is feasible especially in the case of resource constrained Java Cards.

For future work we will focus on the bytecode replacement process of untyped bytecodes required by the defensive Type Separating approach. This transformation is needed to give the JCVM type information needed during run-time to process untyped bytecodes like *pop*. Furthermore, we are working on increasing the number of separated main types so that it is also possible to detect type confusion between *integralData* like *short* and *byte*.

Acknowledgement

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. 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)
2. Barbu, G., Duc, G., Hoogvorst, P.: Java Card Operand Stack:Fault Attacks, Combined Attacks and Countermeasures. In: Prouff, E. (ed.) Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, vol. 7079, pp. 297–313. Springer Berlin Heidelberg (2011)

3. 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.) Smart Card Research and Advanced Application, Lecture Notes in Computer Science, vol. 6035, pp. 148–163. Springer Berlin Heidelberg (2010)
4. Barthe, G., Dufay, G., Jakubiec, L., de Sousa, S.: A Formal Correspondence between Offensive and Defensive Javacard Virtual Machines. In: Cortesi, A. (ed.) Verification, Model Checking, and Abstract Interpretation, Lecture Notes in Computer Science, vol. 2294, pp. 325–328. Springer Berlin / Heidelberg (2002)
5. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.L.: Combined Software and Hardware Attacks on the Java Card Control Flow. In: Prouff, E. (ed.) Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, vol. 7079, pp. 283–296. Springer Berlin Heidelberg (2011)
6. IEEE: Open SystemC Language Reference Manual IEEE Std 1666-2005, IEEE
7. Iguchi-Cartigny, J., Lanet, J.L.: Developing a Trojan applets in a smart card. Journal in Computer Virology 6, 343–351 (2010)
8. Krieg, A., Grinschgl, J., Steger, C., Weiss, R., Haid, J.: A Side Channel Attack Countermeasure using System-On-Chip Power Profile Scrambling. In: On-Line Testing Symposium (IOLTS), 2011 IEEE 17th International. pp. 222 –227 (July 2011)
9. Leroy, X.: Java Bytecode Verification: An Overview. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 2102, pp. 265–285. Springer Berlin Heidelberg (2001)
10. Mostowski, W., Poll, E.: Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In: Grimaud, G., Standaert, F.X. (eds.) Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, vol. 5189, pp. 1–16. Springer Berlin / Heidelberg (2008)
11. Oracle: Runtime Environment Specification. Java Card Platform, Version 3.0.4, Classic Edition (2011)
12. Oracle: Virtual Machine Specification. Java Card Platform, Version 3.0.4, Classic Edition (2011)
13. Sere, A., 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., Slezak, D. (eds.) Future Generation Information Technology, Lecture Notes in Computer Science, vol. 6485, pp. 459–468. Springer Berlin / Heidelberg (2010)
14. Sere, A., Iguchi-Cartigny, J., Lanet, J.L.: Evaluation of Countermeasures Against Fault Attacks on Smart Cards. International Journal of Security and Its Applications, Vol.5 No.2 pp. 49–61 (April 2011)
15. Sun Microsystems Inc.: Java Card 2.2 Off-card Verifier. White Paper (June 2002)
16. Vertanen, O.: Java Type Confusion and Fault Attacks. In: Breveglieri, L., Koren, I., Naccache, D., Seifert, J.P. (eds.) Fault Diagnosis and Tolerance in Cryptography, Lecture Notes in Computer Science, vol. 4236, pp. 237–251. Springer Berlin / Heidelberg (2006)
17. Vetillard, E., Ferrari, A.: Combined Attacks and Countermeasures. In: Gollmann, D., Lanet, J.L., Iguchi-Cartigny, J. (eds.) Smart Card Research and Advanced Application, Lecture Notes in Computer Science, vol. 6035, pp. 133–147. Springer Berlin Heidelberg (2010)
18. Witteman, M.: Advances in Smartcard Security. Information Security Bulletin pp. 11–22 (July 2002)
19. Witteman, M.: Java Card Security. Information Security Bulletin pp. 291–298 (July 2003)