Automatic source code transformations for strengthening practical security of smart card applications

Vašek Lorenc, Tobiáš Smolka and Petr Švenda

valor@ics.muni.cz,{xsmolka,svenda}@fi.muni.cz

Masaryk University, Brno, Czech Republic

Abstract

The availability of programmable cryptographic smart cards provides possibility to run application in significantly more secured environment then ordinary personal computer. Smart card platforms like Java Card or .NET allow to implement portable applications that can be run on different smart card hardware. Barriers for a skilled Java developer switching to the Java Card platform are relatively small – working applets can be written quickly. Unfortunately, the resulting overall security of the applet is strongly dependent on the implementation of the smart card operating system, related libraries, as well as physical resistance and information leakage of the underlaying hardware. Same Java Card applet may run securely on one smart card hardware platform, but be vulnerable on an other. Defenses implementable on the source code level for later case might exist, but such a situation is unfavorable for applet developer as multiple versions of applet must be maintained to support a wider range of smart cards (although all providing Java Card platform).

In this paper we describe several practical attacks on modern smart cards, discuss possible defenses and propose a general framework for automatic replacement of vulnerable operations by safe equivalents. A code strengthening constructions can be also automatically inserted. Only one version of the applet is maintained for multiple different smart card hardware and personalization of source code is performed in in automated fashion. Practical implementation and examples of usage are presented and discussed.

1 Introduction

The cryptographic smart cards are currently in ubiquitous usage in many areas of daily life starting from mobile phones SIM modules over banking cards up to document signing. A common cryptographic smart card consists from several main components. The main processor (8-32 bits) is capable of execution of ordinary code either in form of native assembler code or more directly bytecode for Java Card smart cards. The cryptographic co-processor is designed to significantly speed up execution of time-consuming cryptographic algorithms (e.g., RSA or DES) and to provide additional protection for cryptographic material in use.

Although significantly more secure then ordinary the PC platform, cryptographic smart cards cannot be assumed as a completely secure environment. Several classes of attacks were developed in recent years, targeting everything on smart card platform from insufficient physical security over side channel leakage to logical errors in implementation. Epoxied packaging is etched-out or small holes are drilled so that micro-probes can be inserted and used to read out the memory content. Power consumption or electromagnetic emanations of the chip is measured with high precision and processed key material is revealed. Faults are induced into the memory so computation is corrupted and may result in unwanted behavior of the applet. Incorrect or incomplete implementation of programming interface can be misused to reveal sensitive data.

A defense can be usually developed and implemented against the particular attack. However, originally simple code may become complicated by that and non-trivial knowledge from developer is also required. Moreover, manual implementation of defense can introduce new coding errors. Additionally, different smart card hardware or operating system implementation may require different defenses. Original idea of "Implement once, run everywhere" is then hard to fulfill.

Our contribution here is a design and prototype implementation of system, which allows to parse applet source code and produce transformed version with incorporated protections. Such automatic transformation provides possibility to maintain only single version of the main applet with all defenses (possibly specific for particular smart card hardware) consistently added later. Transformation rules can be pro-

vided by independent security laboratories and therefore decreasing requirements on developer when defenses and best practices are implemented.

Our work is motivated by following targets:

- To enable clear code development with minimization of the logical errors while defensive code constructions can be added automatically later.
- To incorporate software-level defenses implemented by others and support transparent code reuse without introduction of unintentional coding errors.
- To provide software-level defense against smart card vulnerabilities when different smart card hardware platform without such vulnerabilities cannot be used.
- To support quick reactions on newly discovered vulnerable operations, when immediate switch to other hardware platform is not possible and applet rewriting may introduce new implementation errors.
- To detect potential problems or unintentional consequences of operations used in source code.

The rest of the paper is organized as follows: Section two gives description of selected problems of smart card security, describes testing setup used by us and discuss relevant work in area. In Section three, general framework for automatic replacement of vulnerable operations is proposed. Our prototype implementation is described in section four, together with four practical examples and limitations of the approach. Section five describes potential areas of the future research and concludes the paper.

2 Selected problems of cryptographic smart cards

Besides the programming language, API and libraries, a Java Card programmer has to deal with many obstacles in order to produce robust, functional and secure applet. In order to illustrate some difficulties and attacks, we are going to describe selected problems of the platform.

We will focus on Java Card platform, but described problems generally hold for other smart card platform as well – namely .NET or MULTOS smart cards. Underlaying hardware is usually the same or very similar (main CPU, EEPROM memories, cryptographic co-processor) and basic principles of software environment as well (applet isolation, memory management, available libraries). Therefore, described attacks are usually relevant also for other platforms.

2.1 Java Card platform security problems

With the complexity of Java Card programming topics, more than just programming language related problems should be taken into account — issues coming from the long and sometimes unclear platform specification, multi applet environment and inconsistencies on the platform implementation level might lead to hidden security risks, investigated more in [3], [12], [2].

To an accidental observer, Java Card platform looks like a full featured Java environment — there are plenty of properties shared by both the platforms. E.g. Java language and bytecode is used in Java Card, although the second one implements only a subset of the whole ("desktop") Java specification, mainly due to the limited resources of the platform. Unfortunately, there are some substantial differences that can become security benefits or be source of security problems:

```
public class OwnerPIN implements PIN {
  byte[] triesLeft = new byte[1];
  byte[] temps = JCSystem.makeTransientByteArray(1,
      JCSystem.CLEAR_ON_RESET); // temporary array

  boolean check(...) {
    ...
    // compute new value of the counter:
    temps[0] = triesLeft[0] - 1;
    // update the try counter non-atomically:
    Util.arrayCopyNonAtomic(temps, 0, triesLeft, 0, 1);
   ...
  }
}
```

Figure 1: PIN verification function resistant to transaction rollback [3]

No dynamic class loading and threading can reduce the complexity of reasoning about security properties of an applet. Limited support for threading was introduced recently in Java Card 3.0 Connected Edition specification.

Applet firewall mechanism is a key element in controlling and restricting data flow between different applets installed on one smart card. Improper implementation of this system compoment could lead to private data leakage and/or modification.

Atomic operations and transactions may help the programmer to keep the applet data consistent regardless of the card tears.

Despite its useful properties, transactions could arrange hardly predictable results when improperly combined with a code that was not designed with transactions on author's mind. Moreover, even the Java Card specification prior to version 2.0 had a reference code of PIN verification function prone to transaction rollback. If a less careful programmer would use it within a running transaction, it would leave to an attacker an unlimited number of PIN guesses. A better version (Figure 1, explained in [3]) uses non-atomic functions to modify a PIN counter value.

No on-card bytecode verifier is a critical missing component when preventing mistyped code to access memory beyond outside. Although there are some cards with on-card verifier, the verifier itself can be limited and incomplete, making relying on such verification unsafe [3].

Bytecode verification is supposed to prevent attackers from viewing and modification of different memory locations in Java environment, where types and memory accesses are usually strictly controlled. Problem with Java Card platform lies in very limited resources, so that on-card verification is usually unfeasible and thus *off-card verification* has to take place instead. Despite the fact that the off-card verification might prevent some problems, it is clearly inefficient against attacks focused on malicious bytecode modifications after the verification process and faults induced during the applet execution.

Applet/data persistence might be a source of an unexpected problems when programmer does not use a proper variable initialization — variable initialized only in one applet run might be misused when dealing with card tears and consequent runs.

Absence of garbage collector could make memory (de)allocations more error-prone, especially when combined with transaction mechanism. Even a code perfectly correct from the verifier perspective may eventually lead to an invalid memory access, causing either negative impact for the applet environment or allowing the attacker to read out all of the available memory of the smart card [4].

Many Java Card Runtime Environment (JCRE) issues arising from small but subtle differences between Java Card and Java environments might illustrate the complexity of programming Java Card applets. This is the situation when high-level approach might hide important differences causing the security problems for resulting Java Card applet.

2.2 Multi Applet Programming

Developing of a single applet for Java Card might force a programmer to overcome some difficulties, either technical or security ones. However, design and development of an applet that has to share the same card and some private data with other applets might be even more challenging.

Difficulties arise from the fact that other applets developed typically by third party will occupy the same card and could share some data in some instances. Malicious applets installed on the card might be used to build a power-analysis profile of the card, overcome Java Card runtime checks and/or read out all available memory in order to retrieve private data.

Besides these, difficulties with *Shareable Interface Object* became another source of possible security issues.

On Java Card platform, every applet and system runtime has its *security context* that has to be checked by the runtime environment when objects and applets try to communicate and share data. Applet firewall uses object-oriented approach to control behaviour and data visibility between different applets.

In [12] authors demonstrated that some implementations of Java Card firewall are vulnerable to inappropriate manipulation with *Shareable Interface Object*, revealing sensitive information about system objects with *isinstance()* or *equals()* functions and in the worst case leaving a chance to manipulate system AID, making some of the protection mechanisms in Java Card inefficient.

2.3 Power analysis

A significant thread to smart card hardware comes from the family of side channel attacks. The attention to the power analysis was first drawn by Kocher et al. [5]. They presented a powerful power analysis attack, differential power analysis, which led to the extraction of the DES keys used by the smart card. Since then, many researchers has focused on the power analysis attacks and many new techniques have been developed. Lot of effort has been spent on the power analysis of ciphers, such as DES and AES. An exhaustive overview of the power analysis techniques and countermeasures can be found in [7]. Although original attack is more than ten years old, defense mechanisms implemented by smart card manufacturers are still far from perfect.

The power analysis enables an attacker to obtain potentially sensitive information on the operations executed by the smart card as well as on the data processed. It is based on measurements of the current drawn by the card from the reader. The measurements can be done with low cost devices using a small resistor (e.g. 30 Ω) connected in series to the smart card ground line and a digital oscilloscope. The oscilloscope is attached across the resistor and measures the fluctuation of current. The fluctuation forms a power trace (see Figure 4 for example), which displays the current consumption in time. The typical smart card power consumption varies in the order of single milliampere to tens of milliamperes.

The main focus of existing countermeasures is on the protection of the cryptographic material during execution of cryptographic algorithm - e.g., DES, AES or RSA computation. Such algorithms are executed in special cryptographic co-processors with increased protection. Protection is partially based on the obscurity and the exact protection mechanisms are usually not published by smart card manufacturer. Obtaining cryptographic material via power analysis of cryptographic algorithm from cryptographic co-processor is difficult for an attacker today. However, other parts of the code execution on smart card received much smaller amount of attention. Not only cryptographic co-processor, but also execution of instructions in the main processor should be protected, as potentially sensitive data might be processed also on this level. In the seminal paper of this field [9] the authors exploited the power analysis techniques to reconstruct the bytecode of the Java Card applet running on the smart card.

Our power analysis platform, SCSAT04 (see Figure 2), was developed in collaboration with VUT Brno. It integrates multiple side channel tools on a single board. The core is an embedded linux running on the etrax CPU, which provides communication with PC via ethernet network and with a smart card via an integrated smart card reader. The board also contains 200MHz oscilloscope and 12bit A/D converter for measuring the smart card power consumption. Beside power analysis, the SCSAT04 is capable of fault induction via peaks on smart card power supply, data and clock bus. At the current setup, we are able to sample the card power consumption with the digitalization frequency up to 100Mhz and with the resolution of 12 bits per sample. The SCSAT04 board is equipped with 48MB of fast RAM, thus it can

store up to 24 millions samples. The sampling is triggered on a specific pattern of bytes on the I/O wire. The SCSAT04 board was designed to introduce as little noise as possible into the measurement process.

Figure 2: SCSAT04 power measurement device for cryptographic smart card analysis.

2.4 Constructions vulnerable to power analysis

The sensitive data processed by the smart card might be revealed by power analysis in three different ways. First, the data can be revealed by statistical means directly while they are processed by a vulnerable instruction. Second, if different instructions are executed depending on the sensitive data, an attacker can analyze the program flow and thus reveal the data. And third, if single instruction execution differs depending on the data, the attacker again is able reveal the data.

The first way is the most powerful but also the most difficult one. The vulnerability stems from the fact, that the amount of power consumed by the card during an execution of an instruction depends on the data (usually on its Hamming weight) manipulated by the instruction. This vulnerability has been well studied and number of attacks has been proposed including Kocher's differential power analysis. However this kind of attack is extremely difficult in real conditions with commercial cryptographic smart cards. With our current setup, we have not been able to successfully reveal any information on processed data this way.

The other two ways are more usable. We have shown, that it is possible to identify particular bytecode instructions and reconstruct the bytecode executed [6]. So if the program flow is controlled by the sensitive data, the attacker might be able to reveal them by bytecode reconstruction. Consider an *if-then-else* construction. The attacker is not able to obtain values involved in the condition directly from power trace, however if she can tell the difference between execution of *then* and *else* branch (because sequence of instructions is different), she reveals the result of the condition anyway. Thus if the condition is based lets say on the key bit, she can easily reveal its value. Simple countermeasure is to execute similar bytecode in the both branches. This counters also the time analysis (information leakage about processed data based on time needed to finish the operation), because the time of execution of both branches is same and constant.

Similar problem arises in the cases of *for* and *while* cycles. The attacker is able to determine number of loops executed. But also this can be easily countered. Cycles depending on sensitive values should always perform constant number of similarly looking loops, even though some of them are not necessary. This approach again effectively counters the time analysis as the time of execution is constant. Note that using random number of loops could partially solve the problem, but the attacker might be able to estimate the original value by averaging over multiple runs of the application.

Example of suchlike vulnerability can be found in comparison of arrays. Typical comparison is done sequentially and finishes just after a different element is found. Suppose the application is comparing

arrays with PIN digits in plaintext. Then if the comparison finishes just after the second digit was compared, attacker knows that first digit was correct but the second not. Therefore she can guess the digits separately one at a time and thus rapidly increase her chances. Secure comparison should be constant in time and ideally non-deterministic and not sequential. It means that elements are compared in random order, which is changed on each run of the applet. This can effectively counter also statistical attacks on the processed data.

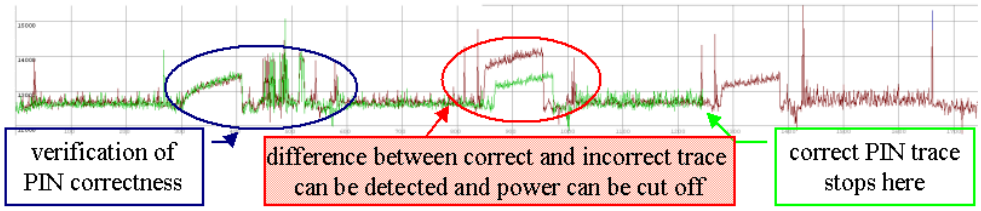


Figure 3: An example of the power trace of PIN verification procedure. Difference between correct and incorrect PIN is clearly visible.

Sensitive information can be also leaked during an evaluation of the boolean expressions. If the lazy evaluation strategy is implemented then an attacker can reveal values of variables used in the expression. Consider a boolean expression $A\&B$. In the case of very quick evaluation, attacker knows that A was *false*, otherwise A was *true*. This expression can be easily fixed by transformation into its equivalent $!(!A\|\|!B)$. Note that non-optimizing compiler should be used to prevent the compilation into vulnerable expression different from non-vulnerable one written in the source code. Result of the transformation should be also verified in the compiled bytecode to confirm, that compiler didn't changed intended representation of the bytecode.

The above examples demonstrate how sensitive values can be extracted just by the analysis of the bytecode executed. We have also presented the defense mechanisms, which effectively counters them. However we have encountered another weakness, which enables the third power analysis technique that might lead to sensitive data leakage. We have found out [6], that appearance of instructions for conditional jump, like *ifeq*[1] or *ifne*[2], differ depending whether the jump was taken or not. The reason probably comes from the way how Instruction Pointer (IP, pointing to the instruction to be executed) is manipulated. If no jump in code is performed, IP is incremented by one via processor native *inc* micro-instruction[3]. If the conditional jump is performed, IP is incremented by the value usually bigger than one (offset to jump target) and therefore *add* micro-instruction is used. As *add* micro-instruction is more complicated than *inc*, its duration is longer and results in significantly different power trace for the parent bytecode instruction. Thus even if the programmer follows secure programming patterns and both branches of the *if-then-else* construction execute similar bytecode, attacker might be still able to reveal the result of the condition.

Fortunately, this *if-then-else* construction can be avoided and replaced by semantically equivalent *switch* construction, which uses *stableswitch* bytecode instruction. The *stableswitch* instruction still leaks some information when first branch from all *case* statements is taken (on some hardware platforms), but not for the other branches. The reason is probably the same as for the *if* instruction – the first branch increments IP only by one (next instruction after *switch*) where other branches require bigger change of IP via *add* micro-instruction. The secure replacement of the *if-then-else* by the *switch* construction is not completely straightforward and is described in the greater details in the section 4.

2.5 Bytecode reverse engineering

Power trace might leak some information on the operations performed by the smart card. In particular, execution of single instructions of the processor can be detected and sometimes even type of the instructions can be identified accurately. Accordingly, we might be able to partially reconstruct (reverse-engineer) source code of an application running on a smart card and obtain potentially sensitive information about processed data. We have tested our approach on fourteen different types of commercially available Java

[1] Instruction jump if equal.
[2] Instruction jump if not equal.
[3] One bytecode instruction comprises from several micro-instructions.

Card smart cards from the leading smart card manufacturers. It turned out, that only a third of them was resistant to this kind of power analysis technique with our measurement setup.

Some cards were easier to attack than others. These cards execute a special pre-instruction before each bytecode instruction. This pre-instruction is clearly visible in the power traces, see the parts marked as JF in the Figure 4). We have called it a "separator", in figures marked as JF, because it effectively separates subsequent instructions. We consider the presence of separators as a vulnerability as it makes the whole process of reverse engineering much easier. The separators are usually easy to find and once you have the instructions isolated, you can directly compare them with the templates from the database.

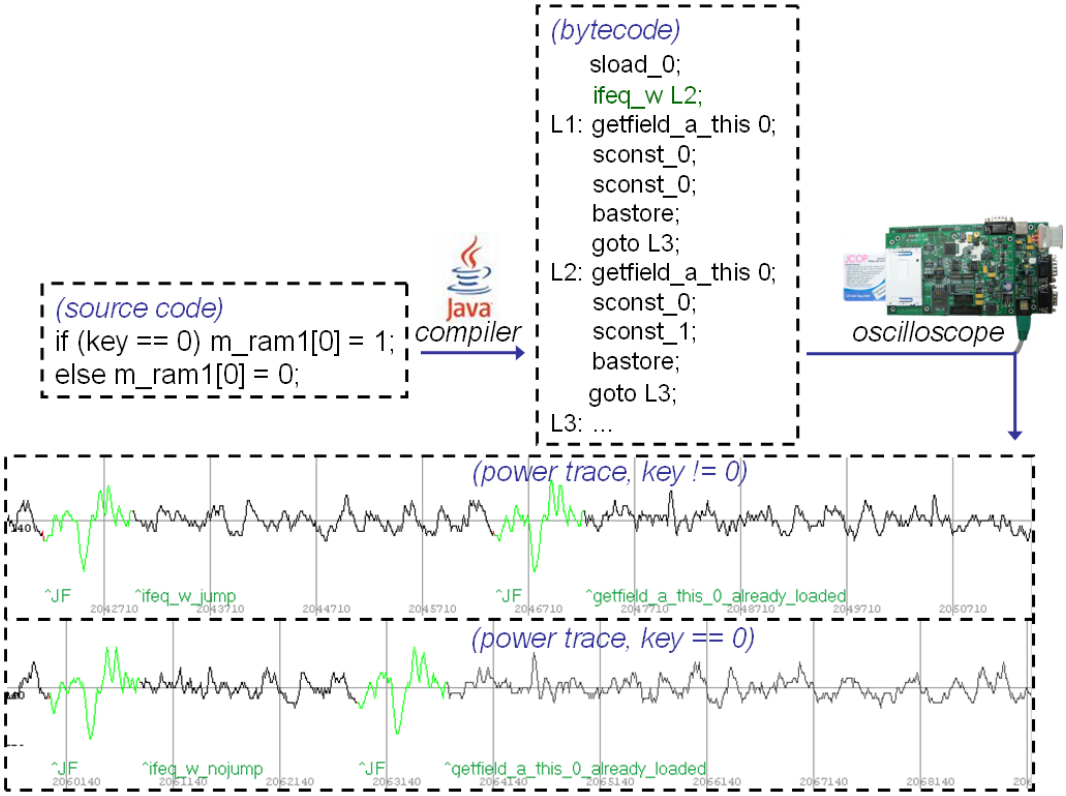


Figure 4: Example of process of reverse engineering of Java Card applet. Bytecode instructions are visible in power trace and allow to distinguish whether then or else branch was taken based only on *if_eq* operation.

The problem of the vulnerable operations described in the previous section can be solved by switching to a different non-vulnerable smart card hardware. Such solution might be appropriate when only small amount of the smart cards was purchased and other suitable hardware platform exists. Yet platform switching might not be an option when particular platform is already in wide use with the significant resources invested or when simply no other suitable platform with required functionality (memory, speed, supported cryptographic algorithms) is available.

During the power analysis of bytecode instructions we observed that not every instruction leaks an information about its operands. Some vulnerable instructions might have semantically equivalent replacement instructions that is not vulnerable to attacks presented above. If such replacement instruction exists, source code of Java Card applet can be modified to use different programmatic structure that will compile into sequence of non-vulnerable instructions instead of vulnerable ones.

3 Automatic replacement framework

General patterns for secure programming in the presence of side channel attacks like [11] were developed and are usually followed by the developers to some extend. It introduces the best practices for the developers of security critical devices with side channel leakages or fault induction vulnerabilities. However, consistent incorporation of best practices remains and issue. Proposed defensive constructions often decrease code clarity, make it less readable and increase probability of introduction of unintentional error.

Demonstration of this fact is an example of the pathway from the naive to the robust implementation of the PIN verification procedure described in [10], increasing code length more then fivefold. Some protections are relevant only for some smart card hardware, posing only unnecessary overhead for other. Some constructions cannot be used at all at particular smart card as hardware lacks the support for required operations.

In contrast, our solution tries to remove the burden of the secure programming patterns from the programmers to some extend.

Several practical security-related requirements should be also fulfilled:

1. Java Card application must be available to functionality and security audit after vulnerable instructions were replaced – as code audit purely on bytecode level is significantly more difficult and lot of additional information (e.g., programmers comments) are lost at this moment, replacement of vulnerable instructions should be done on the source code level.
2. Transformed code should clearly show the new version of the code, the old code (that was replaced) as well as description of the replacement rule that was used for it.
3. Code replacement should be easy to adapt to different smart card hardware with different set of vulnerable instructions. The goal is to write applet source code only once in direct and clean way without any restrictions on potential vulnerability of used instructions. The source code of actually deployed applet is then generated automatically based on proposed replacement framework, personalized for particular smart card hardware, taking into account vulnerable instructions existing on this hardware platform.

3.1 Vulnerable instructions replacement framework

Equipped with previously described problems and requirements for defenses, we can now describe details of proposed *automatic replacement framework*. The whole process requires both manual human analysis and software automated steps. However, once manual steps are performed (e.g., by independent security laboratory), main part of the remaining work can be done in anautomated fashion. The overview of whole process is shown on Figure 5.

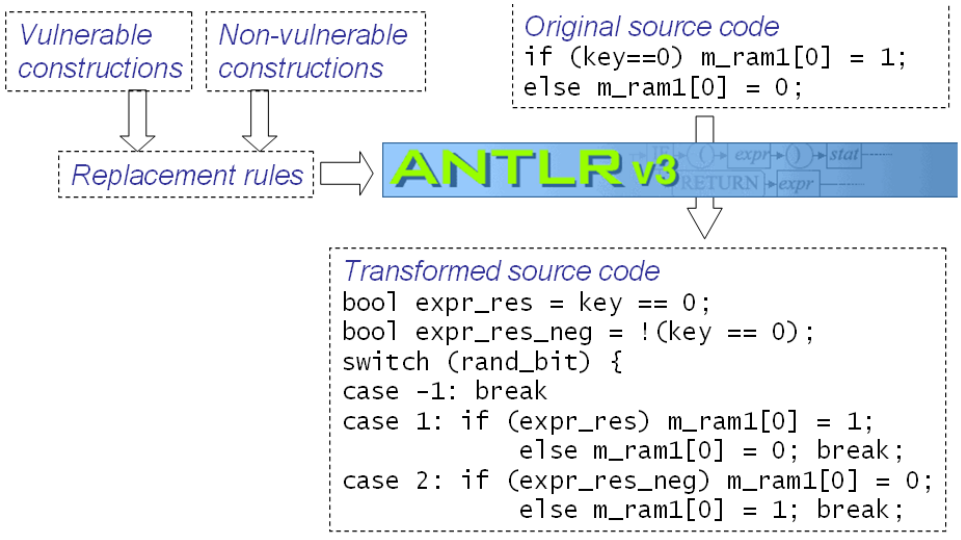


Figure 5: Automatic replacement framework process. Templates of vulnerable constructions and their secure secure replacement are combined in the replacement rules. ANTLR parser is used to automatically transform source code of the applet.

The whole proposed process consists from the following steps:

1. Identification of vulnerable constructions – identification of problems with specific smart card (with testing tools like SCSAT04 or software testing suite [3]) or using well known generic vulnerabilities (e.g., defense against fault induction attack). This step is usually performed once for given smart card hardware and results in list of vulnerable instructions and constructions;

2. Identification of generally vulnerable constructions taken e.g. from the best practices;
3. Identification of the non-vulnerable replacement constructions – results in list of non-vulnerable instructions and constructions that can be used to replace vulnerable ones;
4. Creation of the replacement rules – based on the list of vulnerable and non-vulnerable instruction, semantically equivalent replacement rules are created. See Section 4.1 for example;
5. Processing of the application source code:
   (a) Source code is parsed with syntactic analyzer like ANTLR and syntactic tree is created;
   (b) Code statements to be replaced (vulnerable) are identified, based on the list of vulnerable constructions;
   (c) Replacement rule is found for the vulnerable statement;
   (d) Automatic replacement of vulnerable statement is done on the level of parsed syntactic tree – replaced code is commented out;
   (e) Modified source code is generated from modified syntactic tree.
6. Modified parts of the code can be used as an input for manual audit – transformed code is still in human readable form and contains commented parts of replaced code with identification of used replacement rule;
7. Compilation and analysis of resulting power trace to verify robustness of replaced code.

Whole programmatic constructions can be targeted. The replacement rules can be created to target not only the power analysis leakage (e.g., branch taken in *if_eq*), but also fault induction resiliency (e.g., introduction of shadow variable containing copy of inverted variable value), additional strengthening when correctness of exact implementation of smart card hardware is not known (e.g., additional counter of allowed attempts to PIN verification) or introduction of best practices techniques (e.g., robust clearance of key material before deletion). Examples can be found in Section 4

Special attention should be payed to behavior of compilers as vulnerable sequence of instructions can be still produced due to compiler optimization even when source code was carefully written. Manual analysis of resulting bytecode is vital addition to source code analysis with respect to existence of vulnerable constructions.

4 Prototype implementation

The prototype implementation of described framework was performed. We used freely available yet very powerful parser ANTLR (ANother Tool for Language Recognition)[1] as a tool for parsing Java Card source code (grammar for Java 1.5 was used), manipulating vulnerable statements and producing output source code. ANTLR grammar is used for description of vulnerable statements as well as its non-vulnerable replacement.

4.1 Example: Vulnerable bytecode replacement

As was discussed in Section 2.4, instructions of conditional jump (*if* family – *if_eg, if_neg, ...*) may leak information about the branch selected for program continuation. Such leakage is clearly unwanted as it reveals result of condition evaluation even when both branches are identical on the instruction level (prepared in such way as a defense against timing attack), thus leaking information about variable used in expression and rendering such defense ineffective. Partially non-vulnerable *switch* instruction was identified for the same platform that is not leaking information about the branch taken (see Section 2.4 for discussion).

Equipped with such a knowledge, we can design replacement rule that will automatically replace occurrences of vulnerable *if-then-else* statement to semantically equivalent and secure statement based on *switch* instruction and randomization. The transformation is not completely straightforward. At first, bogus code of branch that will never be used must be added at position just after *switch* instruction in compiled bytecode. This bogus branch will occupy vulnerable position accessible with *inc* micro-instruction (see Section 2.4 for rationale). Additionally, *switch* operates over integer operand where *if* operates over boolean expression. Simple trick to typecast operand using conditional statement $expr?1 : 0$ (where *expr*

is original boolean expression from *if-then-else* statement) cannot be used as it keeps vulnerability in the conditional expression – an attacker can observe the evaluation of the statement $expr?1:0$.

Therefore, randomization needs to must be introduced. Result of expression is evaluated in advance (expr_res variable) as well as its negation (*expr_res_neg*). The actual *switch* branch taken is selected in runtime based on the random variable with two possible values, 0 and 1. If random variable is equal to 0, *switch* branch (*case*) containing original ordering of *if-then-else* branches and expr_res is taken. Otherwise branch with inverted result of boolean expression (*expr_res_neg*) and swapped *then* and *else* branch is taken. This replacement rule is more formally described on Figure 6.

```
// grammar snippet describing IF statement, vulnerable on certain platforms
^(if parenthesizedExpression ifTrue=statement ifFalse=statement?) ->
ifTransformation(
  expr={$parenthesizedExpression.text},
  ifTrue={$ifTrue.text},
  ifFalse={$ifFalse.text}
)

// grammar snipped describing replacement SWITCH construction
ifTransformation(expr,ifTrue,ifFalse,random_bit) ::= <<
boolean exp_res = <exp>;       // result of original expression
boolean exp_res_neq = !<exp>; // inverted result of expression
switch (random_bit){
        case -1:
                // never executed
                break;
        case 1:
                if (exp_res) <ifTrue> else <ifFalse>
                break;
        case 0:
                if (exp_res_neq) <ifFalse> else <ifTrue>
                break;
        default:
                throw new Exception();
}
>>
```

Figure 6: Example replacement rule for (vulnerable) *if-then-else* statement by non-vulnerable construction using *switch* statement with randomized execution of original and inverted command. Slightly simplified for the clarity reasons.

The described replacement will results into non-vulnerable construction on given platform according to the following analysis. An attack should not be able obtain knowledge about expression operand as he cannot:

- Distinguish which *switch* branch (*case*) was taken – the value of random variable is unknown to an attacker and *switch* instruction itself is not leaking information about branch taken. Therefore attacker has no knowledge whether $if(exp\_res) < ifTrue > else < ifFalse >$ or $if(exp\_res\_neq) < ifFalse > else < ifTrue >$ statement will be executed even when source code is known to him.
- Distinguish which *if-then-else* branch was taken – an attacker can still observe execution of *if* instruction and decide whether first (*then*) branch or second (*else*) branch was taken. But as he cannot distinguish whether jump decision was based on *exp_res* or *exp_res_neg* nor he knows *switch* branch, both possibilities are equally probable for him.
- Infer information about operand evaluated in *if-then-else* expression – as branch taken cannot be distinguished, an attacker cannot distinguished whether *ifTrue* or *ifFalse* statement was executed[4].

[4] Assuming that both statements consists from same sequence of instructions.

Ultimately, an attacker cannot infer the information about the operand evaluated in *expr* expression as both results (*expr* is true/false) are equally possible.

Although such a replacement can done manually in principle, resulting code is significantly less readable than original *if-the-else* instruction and is hot candidate for the coding mistakes. Usage of automatic replacement framework will help here to develop straightforward source code and add complicated non-vulnerable construction later.

4.2 Example: Protection against memory fault induction

Another practical example, usable as a defense against wide range of fault induction attacks, is introduction of the shadow variables or shadow arrays to detect faults induced by an attacker in memory. Shadow variable usually contains inverse value of original variable and consistency is checked (resp. updated) every time and original variable is used (resp. changed). Such protection is example of more general protection – is independent from particular hardware and should be widely used. Yet, implementation of such defense obscures significantly source code as variable consistency against shadow counterpart must be ensured.

Replacement rule for this protection was implemented within proposed framework. Developer implements source code without mentioned defense. Source code is then parsed, variables in code are identified and shadow counterparts are automatically created. Special "getter" and "setter" functions are created for every used primitive data type and inserted into original code. "Getter" takes care for verification of variable consistency against shadow variable, "setter" provide variable update functionality. The example of such transformation is shown on Figure 7.

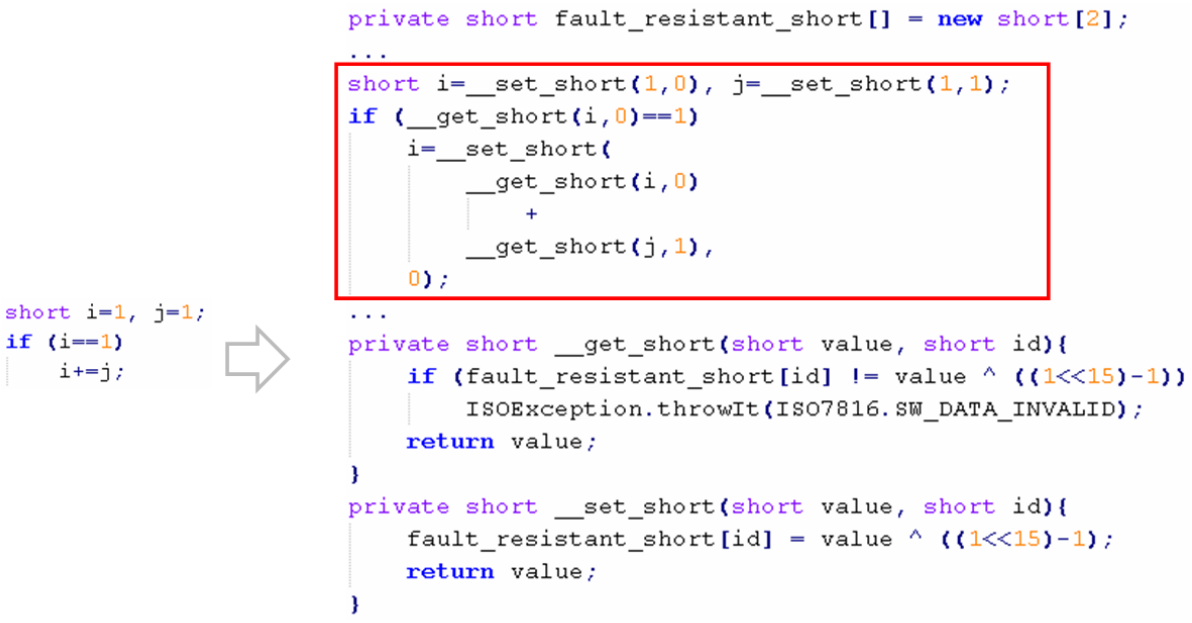


Figure 7: Example of source code transformation adding robust protection against fault induction by creation of shadow variables with an inverse value. Array *fault_resistant_short* contains shadow variables for *short* type. Variable in original code is replaced by "setter" or "getter" call performing lookup into *fault_resistant_short* array and checking consistency.

4.3 Example: Robust state checking and transition with visual modeling

Typical Java Card applet is typically operating in several logical states with different levels of authorization. Some information like applet identification number might be accessible to anybody. Possibility to use to use on-card signature key is available only after user was authenticated by PIN. Blocked user PIN can be unblocked only when administrator authenticated with his own secret key. Usual solution is to check fulfillment of security preconditions before operation is executed (e.g., OwnerPIN::isVerified()

before signature key is used) or special variable holding current logical state state (e.g., "user authenticated" or "admin authenticated") is introduced and checked. With the exception of very simple applets, maintaining a correct checks of logical states, especially when sudden card removal from reader may occur and functions are later added to source code of existing applet.

Two main categories of problems with state may occur:

1. Bug in code – improperly verified transition between states or unintended function call without fulfillment of all security preconditions. Function call may be accessible to outside attacker by sequence of operations unintended by developer.

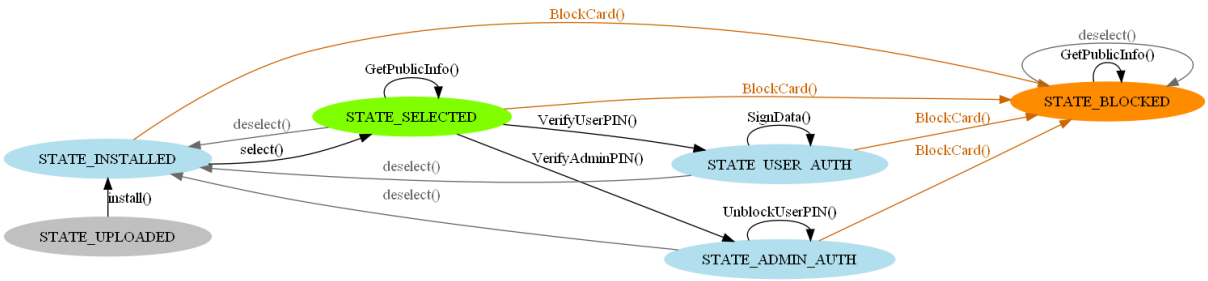
2. Fault attack – Applet security state is changed despite the correct code implementation. An attacker may be able to make (usually random and untargeted) change in smart card memory or instruction sequence. If variable holding applet state is corrupted, an attacker may be able to execute sensitive operation without prior authentication. Result of comparison against expected state may be corrupted as well during execution of comparison instruction.

Proposed solution robustly enforcing state checking and transition is based on following steps:

Model and visualization of allowed state transitions and function calls – developer can easily define, change and inspect visually state transition model as well as functions callable in each security state. Model is automatically and consistently transformed and integrated into existing source code. For model description and visualization, we used GraphViz grammar[5] as it can be conveniently created, visualized and inspected. ANTLR grammar was created, allowing to parse GraphViz grammar and transform it into Java Card source code.

State transition enforcement – state transition is always moderated over specialized function and new state can be only one from well defined set of consequent states. Developer only adds SetStateTransition(newState) function call every time an applet likes to change its security state (e.g., after user PIN is verified). Transition is checked against set of allowed transitions and permitted only when transition between *currentState* (hold in special variable) to *newState* exists in the model.

Function independently verifies if can be called – function itself tests, whether can be called in present state and context before continues to execute code in its body. Function VerifyAllowedFunction() is added at beginning of every function. If current function is not allowed in the model to be called in the present security state, applet execution is aborted with appropriate response (e.g., logging or even card blocking).



[5] http://www.graphviz.org/

```
private void SetStateTransition(short newState) throws Exception {
  // CHECK IF TRANSITION IS ALLOWED
  switch (m_currentState) {
    case STATE_UPLOADED: {
        if (newState == STATE_INSTALLED) {m_currentState = STATE_INSTALLED; break;}
        throw new Exception();
    }
    case STATE_INSTALLED: {
        if (newState == STATE_SELECTED) {m_currentState = STATE_SELECTED; break;}
        if (newState == STATE_BLOCKED) {m_currentState = STATE_BLOCKED; break;}
        throw new Exception();
    }

    case STATE_SELECTED: {
        if (newState == STATE_SELECTED) {m_currentState = STATE_SELECTED; break;}
        if (newState == STATE_USER_AUTH) {m_currentState = STATE_USER_AUTH; break;}
        if (newState == STATE_ADMIN_AUTH) {m_currentState = STATE_ADMIN_AUTH; break;}
        if (newState == STATE_BLOCKED) {m_currentState = STATE_BLOCKED; break;}
        if (newState == STATE_INSTALLED) {m_currentState = STATE_INSTALLED; break;}
        throw new Exception();
    }
```

Figure 8: Example of applet transition model visualization and resulting source code for SetStateTransition() function. Similar code is generated also for VerifyAllowedFunction().

4.4 Example: Java Card Firewall Implementation Issues

Designing an applet for smart card where other applets of third-party providers will be installed may put higher demands on proper Java Card firewall implementation.

Altough it might be unfeasible to overcome all non-complient behavior of the firewall of a specific Java Card mentioned earlier in this paper, it still can be useful to know what kind of inconsistencies and problems might appear on the given smart card.

In order to simplify this process, the Java Card Firewall Tester [8] made by Wojcich Mostowski project has been released to public use, consisting of set of applets and host application. Altough some cards cannot be tested due to their limitations (either memory limitations or too restrictive applet verification), the result of this tester is still very useful in the rest cases.

```
Non-external ciphers should not be usable externally.
INS: 0x13 Test#: 13
```

Figure 9: JC Firewall Tester reports non-compliant handling of ciphers declared as non-external

Even if the firewall tester does not detect a real security problem, it still might reveal too restrictive (or too permissive) behavior of the JCRE that could help the author of the applet to change the design in order to make it run or that some specific Java Card properties are not checked properly by the runtime (figure 9).

Another notification for a programmer could be helpful in case of inappropriate use of transaction and non-atomic features of Java Card. In order to illustrate possible problems with (non-)atomic operations, see a code snippet in figure 10. There are obviously two different results for almost identical set of operations, depending on the order of the operations within the transaction — a variable is backed-up just before it is updated conditionally for the first time. Although this is logical behavior, this fact might prepare hard times for an applet designer.

```
a[0] = 0;                              a[0] = 0;
beginTransaction();                    beginTransaction();
  a[0] = 1;                              arrayFillNonAtomic(
  arrayFillNonAtomic(                      a,0,1,2); // a[0] = 2;
    a,0,1,2); // a[0] = 2;              a[0] = 1;
abortTransaction();                    abortTransaction();
// a[0] == 0;                          // a[0] == 2;
```

Figure 10: Atomic and non-atomic update of a variable with different results[3]

Such a situation cannot be automatically solved by the transformation tool. As a possible result of applet rewrite, even the author seems to be unsure whether the variable should be a subject of transaction rollback or not. However, programmers can only be informed that both conditional and non-conditional update of a persistent variable is being requested inside a transaction and that this may lead to unpredictable results and variable states and that more developer's attention is needed.

These symptoms can be detected by our transformation tool, since the combination of started transaction, a local variable updated with assignment statement and functions `arrayCopyNonAtomic()` or `arrayFillNonAtomic()` called on the same variable can be analyzed on the source level.

4.5 Limitation of approach

There are several practical limitation to the described approach that may affect applicability to some extend, depending on the area where automatic code transformation or replacement is used.

At first, non-vulnerable replacement construction needs to be found. This requires careful inspection of power trace of candidate replacement constructions for possible vulnerabilities. E.g., *switch* instruction was used in one of our examples, but if first branch is utilized, replacement construction will be still vulnerable. Therefore, no proof of vulnerability resistance of resulting bytecode is provided (proofs can be provided in theory, if accurate model of power leakage for particular device is known – but that is frequently not the case).

Readability of the transformed code is usually impacted and possibility of introduction of unintentional bugs to code by transformation is opened. Still, it is usually better option to have simpler code written by the developer itself with complex replacement constructions added later. But transformation needs to be tested and audited carefully.

Depending on properties of replacement construction, computation and memory overhead might be introduced. E.g., described *if-then-else* replacement requires more then three-times more instructions to begin execution of the proper branch. If the transformation requires significant amount of duplicated variables, restricted memory available to the applet might be exhausted. The introduction of shadow variables as a protection against fault induction doubles the required memory and requires function call every time a variable is accessed or modified. Nevertheless, current smart cards usually provide enought power and memory to facilitate additional overhead for a typical applet.

Device-dependent transformations (e.g., vulnerable *if-then-else* instruction transformed by *switch*) need to be kept up to date for particular smart card hardware (if different processor is used, vulnerability assessment needs to be redone). More general transformation (e.g., additional shadow variable) requires less maintenance as are independent from particular hardware.

And finally, replacement construction simply not exists sometimes or is too complicated to create generally applicable replacement rule for ANTLR.

5 Conclusions and future work

This paper described practical vulnerabilities of current smart cards against both logical and physical errors. General replacement framework is described as a way how to automatically, consistently and in error-free way introduce protections on source code level against them. The framework takes the source code of Java Card applet and transform it automatically into the more secure version. Applicability of the concept is broad – replacement of vulnerable constructions by non-vulnerable ones based on predefined set of replacement rules can be done, as well as consistent insertion of strengthening code (e.g., fault induction

protections). Prototype tool was implemented based on ANTLR parser with several replacement rules to verify practicality of the described concept.

The main idea is that source code is developed only once and in clean way, so number of the programming errors is minimized and a focus on logical correctness can be made. Automated framework then add more complex security constructions and personalize code to particular smart card hardware, taking into account its vulnerabilities. These manipulations can be used to add protection against fault induction attacks (e.g., shadow variables), introduce techniques from best practices (e.g., additional check for PIN verification retry counter) or add techniques hardening power analysis (e.g., introduction non-determinism into code execution). Replacement is done in a way that still supports manual code audit of the final source code.

So far, we focused on smart cards with Java Card platform. However, the proposed transformation framework with ANTLR is generally applicable to other platforms like MULTOS or .NET as well as inspected vulnerabilities steams mainly from generally valid attack threads rather then from particular programming platform.

References

[1] Another tool for language recognition (antlr). http://www.antlr.org, accessed 27.4.2010.

[2] E. Poll E. Hubbers. Transactions and non-atomic api methods in java card: specification ambiguity and strange implementation behaviours. Available at `http://www.cs.ru.nl/~erikpoll/papers/acna_new.pdf` , accessed 27.4.2010.

[3] Erik Poll Engelbert Hubbers, Wojciech Mostowski. Tearing java cards. In *In Proceedings, e-Smart 2006, Sophia-Antipolis*, 2006.

[4] Jip Hogenboom and Wojciech Mostowski. Full memory attack on a Java Card. In *4th Benelux Workshop on Information and System Security, Proceedings*, Louvain-la-Neuve, Belgium, November 2009. Available at `http://www.dice.ucl.ac.be/crypto/wissec2009/static/13.pdf` , accessed 27.4.2010.

[5] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology - Crypto 99, LNCS 1666*, 1999.

[6] Jiří Kůr, Tobiáš Smolka, and Petr Švenda. Improving resiliency of javacard code against power analysis. In *Proceedings of SantaCrypt '09, Prague*, 2009.

[7] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. Power analysis attacks. 2007. ISBN: 978-0-387-30857-9.

[8] W. Mostowski. Java card firewall tester (jcft). Available at `http://www.cs.ru.nl/~woj/firewalltester/`, accessed 27.4.2010.

[9] D. Vermoen, M. Witteman, and G. N. Gaydadjiev. Reverse engineering java card applets using power analysis. In *WISTP 2007, LNCS 4462*, pages 138–149, 2007.

[10] Eric Vtillard. Jc101-12c: Defending against attacks. 2008. http://javacard.vetilles.com/2008/05/15/jc101-12c-defending-against-attacks/, accessed 27.4.2010.

[11] M. Witteman and M. Oostdijk. Secure application programming in the presence of side channel attacks. In *RSA Conference 2008*, 2008.

[12] Erik Poll Wojciech Mostowski. Testing the java card applet firewall. Technical report, Radboud University Nijmegen, 2007.