Improving Resiliency of JavaCard Code Against Power Analysis

Jiří Kůr Tobiáš Smolka Petr Švenda

{xkur,xsmolka,svenda}@fi.muni.cz

Masaryk University, Brno, Czech Republic

Abstract

The power analysis of smart cards is powerful side channel attack, originally introduced by Kocher et al. [2]. The observation of power consumption of smart card with high precision can be used to extract information about executed code as well as about the processed data. Our work describes ways how to obtain information about bytecode instruction from power trace, identify vulnerable operations leaking information about its operand and propose general framework for automatic replacement of vulnerable operations by safe equivalents. Practical implementation and examples of usage is 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. 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 JavaCard smart cards. The cryptographic coprocessor 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.

The power analysis of smart cards is powerful side channel attack, originally introduced by Kocher et al. [2]. The observation of power consumption of smart card with high precision can be used to extract information about executed code as well as about the processed data. Although original attack is more than ten years old, defense mechanisms implemented by smart card manufacturers are still far from perfect. The main focus is on protection of cryptographic material during execution of cryptographic algorithm - e.g., DES, AES or RSA computation. Even when protection is partially based on obscurity and the exact protection mechanisms are not published, obtaining cryptographic material via power analysis of cryptographic algorithm is reasonably difficult for an attacker today. Yet other parts of the code execution on smart card received much smaller amount of attention. Except cryptographic coprocessor, execution of instructions in the main processor should be protected as well, as potentially sensitive data might be processed also on this level.

Our work is motivated by following targets:

- To provide software-level defense against smart card vulnerabilities when different smart card hardware cannot be selected.
- To offer flexibility for developer with secure portability of applet to different smart card platforms as different smart cards may have different countermeasures implemented and may have different operations vulnerable.
- Quick reactions on newly discovered vulnerable operation, when immediate switch to other hardware platform is not possible and applet rewriting may introduce new implementation errors.

The rest of the paper is organized as follows: Section two gives general overview of power analysis, describes setup used by us and discuss previous work in area. In Section three, visibility of bytecode instructions in power trace is described and security implications are discussed. General framework for automatic replacement of vulnerable operations is proposed in Section four. In Section five our prototype implementation is described. Example application of replacement framework are given in Section six. Section seven describes potential areas of future and concludes the paper.

2 Power analysis

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 3 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. More details on smart card power analysis can be found in [3].

2.1 SCSAT04 description

Our power analysis platform, SCSAT04 (see Figure 1), 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.

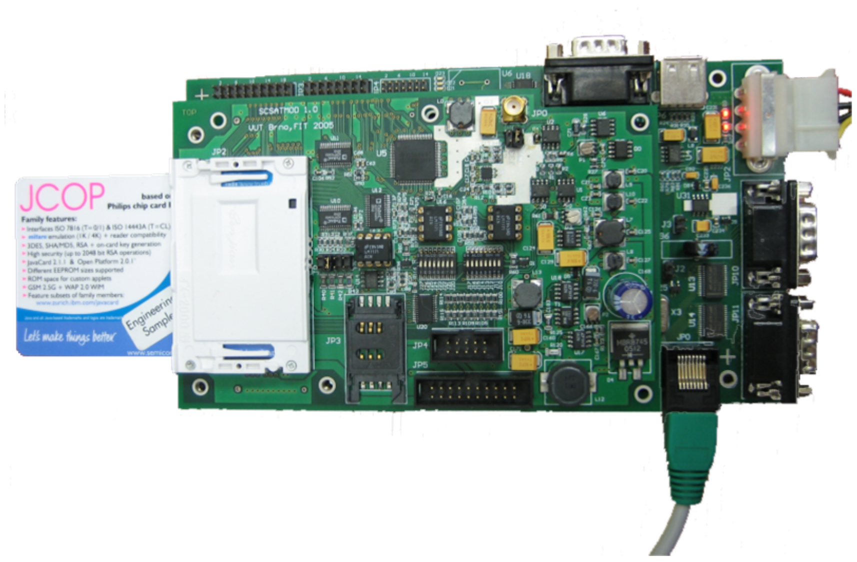


Figure 1: SCSAT04 power measurement device for cryptographic smart card analysis.

2.2 Previous work

The attention to the power analysis was first drawn by Kocher et al. [2]. 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 [3].

Significantly less effort has been spent on the power analysis of instructions of the main processor. In the seminal paper of this field [4] the authors exploited the power analysis techniques to reconstruct the bytecode of the javacard applet running on the smart card. This work is similar to ours, but contains only the bytecode reconstruction and acts as a preliminary step to our main contribution.

Another work related to the main topic of this work [5] proposed general patterns for secure programming in the presence of side channel attacks. It introduces the best practices for the programmers of security critical devices with side channel leakages or fault induction vulnerabilities. In contrast, our solution tries to remove the burden of the secure programming patterns from the programmers to some extend.

3 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 ten different types of commercially available JavaCard 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.

Three 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 3, trace D). 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.

3.1 Instruction database

To successfully reverse engineer an unknown bytecode from target card, an attacker has to have access to a programable smart card of the same type as the attacked one. The process of reverse engineering starts with an identification of a bytecode instructions in the power trace and building a database of instruction templates for particular type of JavaCard smart card. This is done using a programable version of this card, and programming our own testing applet. Knowing the executed bytecode, one might be able to correlate parts of the power trace with the particular bytecode instructions. One possible way to do this, is to start with repeating sequence of instructions. In the Figure 2 a sequence of simple assignments in the source code (A) is compiled into the repeating sequence of two bytecode instructions (B). This repeating sequence is visible in the resulting power trace (trace C) more easily than single instruction. At the bottom of the Figure (trace D) already identified bytecode instruction can be seen, namely *sstore*[1] Once the instruction is identified in the power trace, the corresponding part can be extracted and stored as a template. Quality of this template can be improved by averaging over multiple measurements of the same instruction. Note that the database is build only once for the particular type of a smart card based on known bytecode instructions. Then the database can be repeatedly applied on the same type of smart card to reveal an unknown sequence of bytecode instructions.

Having the database of the instruction templates, one can try to reveal the bytecode of an unknown applet. A power trace is measured, then a similarity search through this trace is performed in order to obtain the sequence of bytecode executed. The instruction templates are taken from database one by one. Using sliding window mechanism, the template is successively compared with all possible parts of the power trace and the similarity is computed. If this similarity reaches predefined threshold, the compared part of the power trace is identified as the searched instruction. The threshold is set by a try&test approach and might differ between different types of smart cards. The metrics for computing similarity are discussed in the following subsection.

[1]*sstore* is a bytecode instruction, that puts an operand of the type *short* on the stack.

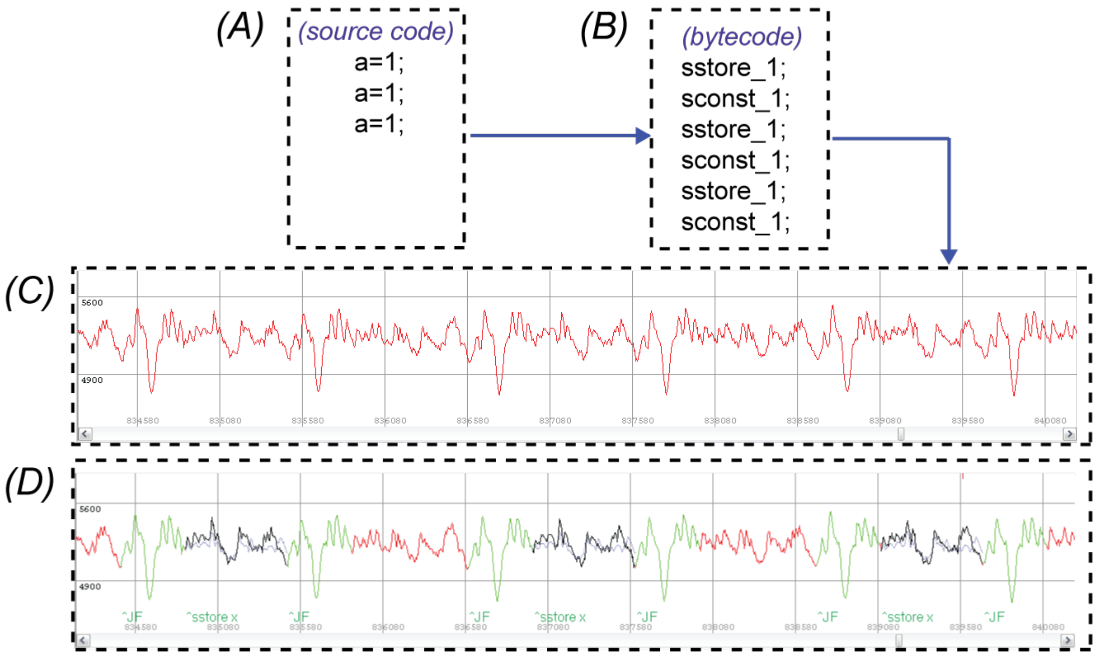


Figure 2: SCSAT04 power measurement device for cryptographic smart card analysis.

3.2 Similarity searching

The similarity searching through the power trace can be tricky. When the same instruction is measured twice, it never produces exactly the same power trace. Although the traces looks similar, they usually suffer from two kinds of deformations. First, they differ in y axis. This deformation is mostly caused by noise during measurement and can be solved by measuring multiple power traces and making an average one. Furthermore, the template matching algorithms are generous to this kind of deformation. Second, the power traces differ in time. This is much harder problem for the template matching algorithms, because corresponding parts cannot be aligned. In fact this time nondeterminism effectively counters also averaging of traces. In practice, the time of an instruction execution varies up to 10 percent of the average time.

We have tested a number of different template matching algorithms, ranging from the simple Euclidean distance of the power trace and the template, through correlation and Haar's wavelets to dynamic time warping. When comparing the performance of the algorithms, we have considered not only an accuracy, but also a speed. The speed of the algorithm is important for its practical usability. In our condition, we need to perform a search of a single instruction in order of minutes. Given the length of the power trace, which is in order of millions of samples and the length of an instruction, which is in order of hundreds or thousands, the complexity of an algorithm matters.

Our experiments have shown that most accurate algorithm is dynamic time warping. It was designed for the very similar problems, when compared vectors slightly vary in time. However it also appeared, that it is extremely slow and thus inappropriate for our research. At the end we have chosen a simple Euclidean distance as our preferred template matching algorithm. It is very fast and the accuracy is comparable to more sophisticated methods, which in turn are much slower.

3.3 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. 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, because the time of execution of both branches is same and constant.

Similar problem arises in 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.

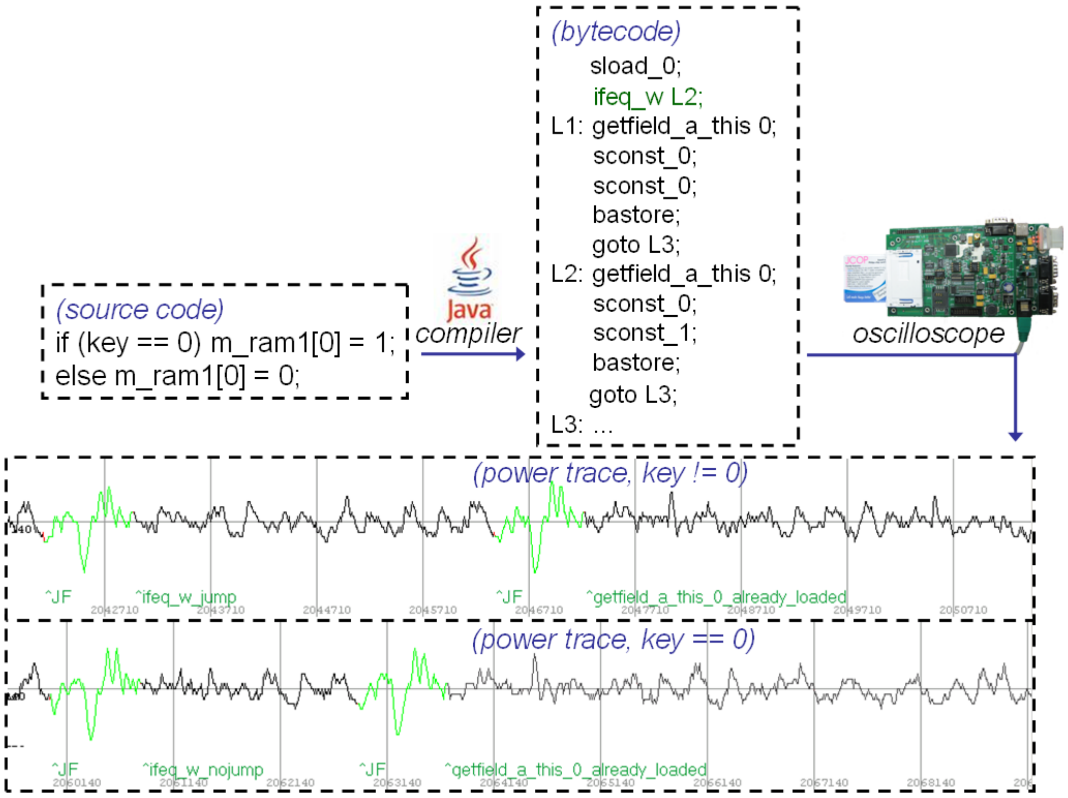
Example of suchlike vulnerability can be found in comparison of arrays. Typical comparison is done sequentially and finishes just after a different element is found. Suppose the application is comparing arrays with PIN digits in plaintext. Then if the comparison finishes just after the second digit was 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.

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 source code. Result of transformation should also be verified in the compiled bytecode.

The above examples demonstrates how sensitive values can be extracted by analysis of the bytecode executed. We have also presented the best practises, which effectively counters majority of 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, that appearance of instructions for conditional jump, like *ifeq*[2] or *ifne*[3], differs depending whether the jump was taken or not. The reason probably comes from the way how Instruction Pointer (IP, pointing to instruction to be executed) is manipulated. If no jump in code is performed, IP is incremented by one via processor native *inc* micro-instruction[4]. 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, but not for 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

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

replacement of the *if-then-else* by the *switch* construction is not straightforward and is described in the following section.



```
(source code)
if (key == 0) m_ram1[0] = 1;
else m_ram1[0] = 0;
```

```
(bytecode)
     sload_0;
     ifeq_w L2;
L1: getfield_a_this 0;
     sconst_0;
     sconst_0;
     bastore;
     goto L3;
L2: getfield_a_this 0;
     sconst_0;
     sconst_1;
     bastore;
     goto L3;
L3: ...
```

Figure 3: Example of process of reverse engineering of JavaCard 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.

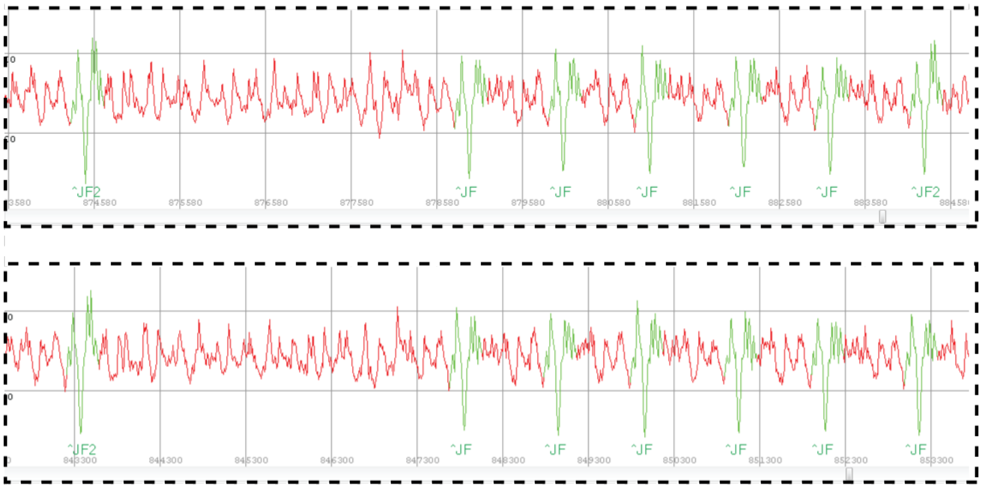


Figure 4: Power traces of secure bytecode construction. Instruction *Ifeq_jump* was replaced by *stableswitch*. Jump is always executed and both branches appear similar.

4 Automatic replacement framework

The problem of vulnerable operations described in the previous section can be solved by switching to different non-vulnerable smart card hardware. Such solution might be appropriate when only small amount of smart cards was purchased and other suitable platform exists. Yet platform switching might not be an option when particular platform is already in wide use with significant resources invested or when simply no other suitable platform with required functionality (memory, speed, supported cryptographic algorithms) is available. Our experiments with fourteen different smart cards from four different manufacturer currently available on the market confirmed that very few of them are not vulnerable to attacks presented above with our measurement setup.

During the power analysis of bytecode instructions we observed that not every instruction leaks 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 JavaCard applet can be modified to use different programmatic structure that will compile into sequence of non-vulnerable instructions instead of vulnerable ones. An example of such replacement was already presented in Section 3.3 where conditional branch realized by vulnerable instruction *if* (on particular card) can be replaced by appropriately formed *switch* instruction. Ideally, such replacement would be automatically executed – automatic replacement framework proposed here is based on this idea. Proposed solution is therefore another method how to mitigate the risk from power analysis of bytecode instructions than switching to non-vulnerable smart card hardware.

The proposed solution is based on following assumptions:

1. List of vulnerable and non-vulnerable instructions is available – as result from detailed power analysis. Manual analysis for each smart card hardware is necessary.

2. Safe equivalents to vulnerable instructions exist – Semantically equivalent instructions (e.g, *if_eq* and *if_neq*) or constructions (e.g., *if-then-else* $\rightarrow$ *switch*) are available to replace identified vulnerable instructions. Replacement of vulnerable instructions must be possible at least on bytecode but ideally on the source code level as well (see rationale below).

Several practical security-related requirements should be also fulfilled:

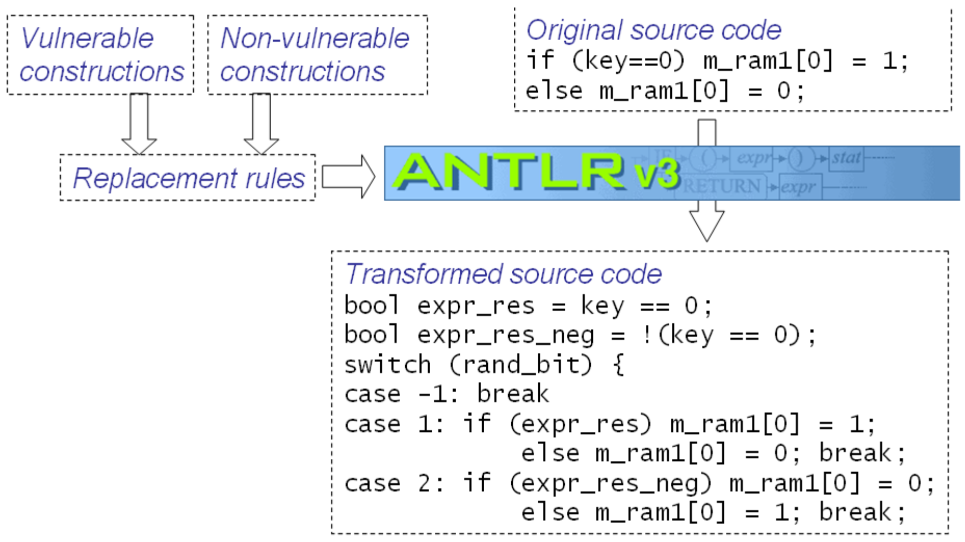
1. JavaCard application must be available to functionality and security audit after vulnerable instructions were replaced – as code audit purely on bytecode level is significantly more difficult and lot of additional information (e.g., programmers comments) are lost at this moment, replacement of vulnerable 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.

4.1 Vulnerable instructions replacement framework

Equipped with previous assumptions and requirements, we can now describe details of proposed *automatic replacement framework*. The whole process requires both manual human analysis and software automated steps. The overview of whole process is shown on Figure 5.



```
if (key==0) m_ram1[0] = 1;
else m_ram1[0] = 0;
```

```
bool expr_res = key == 0;
bool expr_res_neg = !(key == 0);
switch (rand_bit) {
case -1: break
case 1: if (expr_res) m_ram1[0] = 1;
              else m_ram1[0] = 0; break;
case 2: if (expr_res_neg) m_ram1[0] = 0;
              else m_ram1[0] = 1; break;
```

Figure 5: Automatic replacement framework process.

The whole proposed process consists from the following steps:

1. Identification of vulnerable constructions – identification of problems with specific smart card (via SCSAT04) or using well known generic vulnerabilities. This step is usually performed once for given smart card hardware and results in list of vulnerable instructions and constructions.
2. Identification of non-vulnerable replacement constructions – results in list of non-vulnerable instructions and constructions that can be used to replace vulnerable ones.
3. Creation of replacement rules – based on the list of vulnerable and non-vulnerable instruction, semantically equivalent replacement rules are created. See Section 5.1 for example.
4. Processing of 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
5. 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
6. Compilation and analysis of resulting power trace to verify robustness of replaced code

In this paper, we focus mainly on vulnerability of bytecode instructions. But the described process is not limited to such usage only. Whole programmatic constructions can be targeted as well as 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 5.1

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.

5 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 JavaCard 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.

5.1 Example of replacement rule

As was discussed in Section 3.3, instructions of conditional jump (*if* family – *if_eg, if_neg, ...*) often leaks information about branch selected for program continuation. Such leakage is clearly unwanted as 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 3.3 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 3.3 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 5.1.

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[5]. 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.

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

```
// 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.

5.2 Limitation of approach

There are several practical limitation to the described approach that may affect applicability to some extend.

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 our example, 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 instruction to start execution of the proper branch. If transformation requires significant amount of duplicated variables, restricted memory available to applet might be exhausted.

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.

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

6 Conclusions and future work

This paper described practical vulnerability of current smart cards based on power analysis of JavaCard bytecode instructions. General replacement framework is described that takes the source code of JavaCard applet and automatically replace vulnerable constructions by non-vulnerable ones based on predefined set of replacement rules. Replacement rules are created based on detailed analysis of power traces for JavaCard bytecode instructions. Prototype tool was implemented based on ANTLR parser to verify practical usability of described concept.

The main idea is that source code is developed only once and in clean way, so number of programming errors is minimized and 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. Replacement is done in a way that supports manual code audit of the final source code.

The proposed framework can be also used to automatically manipulate source code to introduce protection constructions that makes source code less readable and therefore more error prone when performed by developer. 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).

So far, we focused on smart cards with JavaCard 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 smart card processor and not from smart card operating system and programming platform. Preliminary studies confirmed vulnerability of .NET code executed on smart card (vulnerable *if-then-else* instruction) and we therefore expect that adaptation of proposed framework to .NET platform will also yield practically useful results.

References

[1] Another tool for language recognition (antlr). In *http://www.antlr.org*.

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

[3] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. Power analysis attacks. 2007.

[4] 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.

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