Reverse Engineering Java Card Applets Using Power Analysis

Dennis Vermoen[1,2], Marc Witteman[2], and Georgi N. Gaydadjiev[1]

[1] Computer Engineering, TU Delft, The Netherlands
`georgi@ce.et.tudelft.nl`
`http://ce.et.tudelft.nl/~georgi`
[2] Riscure BV, The Netherlands
{`vermoen,witteman`}`@riscure.com`
`http://www.riscure.com`

Abstract. Power analysis on smart cards is widely used to obtain information about implemented cryptographic algorithms. We propose similar methodology for Java Card applets reverse engineering. Because power analysis alone does not provide enough information, we refine our methodology by involving additional information sources. Issues like distinguishing between bytecodes performing similar tasks and reverse engineering of conditional branches and nested loops are also addressed. The proposed methodology is applied to a commercially available Java Card smart card and the results are reported. We conclude that our augmented power analysis can be successfully used to acquire information about the bytecodes executed on a Java Card smart card.

1 Introduction

Currently Java Card is the most commonly used platform for commercial smart cards. According to Sun Microsystems, Java Card technology grew from 750 million deployments in November 2004 to over 1.25 billion deployments in November 2005 [1,2]. Because smart cards are typically used in applications that require a high degree of security, it is needless to say that security of Java Card applications is very important.

Power analysis is a side channel analysis technique to acquire information about running processes on a device (such as smart cards) by monitoring the dynamic current usage. Power analysis on smart cards is commonly used to obtain information about running cryptographic algorithms [3,4,5,6].

In this paper, we introduce Java Card reverse engineering methodology by means of augmented power analysis. When a Java Card applet source code could be reverse engineered, possible vulnerabilities can be exploited. We performed our experiments on several commercially available Java Card smart cards. In this paper we will focus on only one specific smart card[1]. Experimental results for different smart cards can be found in [7]. Nevertheless, the majority of the proposed techniques are applicable in the general case.

[1] The specific brand and type of the smart card can not be disclosed.

The main contributions of this paper are:

- A methodology to analyse power consumption of Java Card applets;
- Techniques to determine a unique power profile template for each Java Card bytecode. In addition, we describe how templates can be recognised in an arbitrary power trace, in order to determine the execution trace;
- Additional information sources that can be used to reduce the number of errors in the generated trace;
- Techniques to convert the execution trace into structured Java Card bytecode source.

Due to the space limitation, readers are assumed to have some basic knowledge of Java Card technology (a good introduction can be found in [8,9]).

The rest of this paper is organised as follows. Section 2 discusses the methodology that we used. Section 3 presents the experimental results and the methodology refinements. Finally, we conclude in Section 4.

2 Methodology

In order to gain information and reverse engineer arbitrary Java Card applets, we selected a programmable Java Card smart card. A Java Card applet is compiled to bytecode using the Java compiler. For example, each addition operation as depicted in Figure 1 is compiled to the following bytecode sequence

`sload`, `sload`, `sadd`, `s2b`, `sstore`

The multiplication sequence looks similar (i.e. the `sadd` bytecode is replaced by the `smul` bytecode). Therefore, the power trace representing the power consumption variations of the applet execution is expected to show repetitions, making this applet interesting for power analysis.

```
public class TestApplet extends javacard.framework.Applet {
   public void process(javacard.framework.APDU apdu) {
      byte a = (byte) 0x04, d, p;
      byte buffer[] = apdu.getBuffer();
      short len = apdu.setIncomingAndReceive();
      d = buffer[(short)(javacard.framework.ISO7816.OFFSET_CDATA)];
      p = (byte)(a+d);
      p = (byte)(a*d);
      p = (byte)(a*d);
      p = (byte)(a+d);
      p = (byte)(a*d);
      p = (byte)(a+d);
      p = (byte)(a+d);
      p = (byte)(a*d);
   }
}
```

Fig. 1. Example Java Card applet

We can execute the above `process` method by sending an arbitrary command to the smart card when the applet is active. Our acquisition framework, which is described in Appendix A, is used to obtain a power trace from the execution of this test applet. The resulting power trace is depicted in Figure 2. This measurement was performed without any trigger delay, at low speed and the at maximal number of samples possible for the used equipment to gain a complete overview of the smart card power consumption.

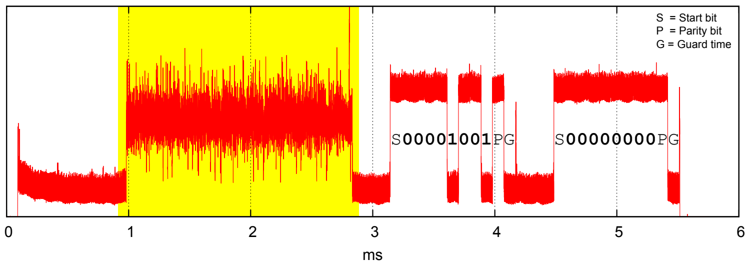


Fig. 2. Single power trace

The last part of the power trace (i.e. from 3 to 6 ms) represents the smart card response. In this case the response was `0x9000`, because the Java Card applet executed successfully. The `0x9000` response code is returned after approximately 3 ms. Therefore, the execution of the actual Java Card applet takes place in the first part of the power trace (i.e. from 1 to 3ms). Note that the power consumption increases and looks noisier during the applet execution. Possibly the investigated smart card activates some countermeasure against power analysis when the Java Card Virtual Machine (JCVM) is running. After we determined the approximate start time and duration of the Java Card applet, a larger number of power traces could be collected. By delaying the trigger signal and decreasing the number of samples, we limited the acquisition to only the interesting region of the power trace (i.e. from 1 to 3ms).

Resampling. Performing power analysis often requires the collection of a significant number of traces. When capturing 10000 traces at 200 MHz each containing 1000000 8-bit samples, the total file size becomes $\approx$ 9.5 GB. Resampling is a technique to reduce the total file size, at cost of losing some information. When the trace set is resampled at 4 MHz (the operating frequency of the smart card), the number of samples will be reduced by a factor 50. Each trace will then contain only 20000 samples [2] and require only 760 MB. Therefore it is advantageous to resample the traces before storing them. Some measurements that require high precision must of course not be resampled as will be shown later.

[2] All samples represent the average value of 50 samples in the original trace.

Correlation. Correlation gives a measure of association between variables [10]. It returns a value between -1 and 1, where 1 means "identical in shape" and -1 means a "inverted in shape". Correlation 0 means that the values are uncorrelated. We use correlation to recognise specific templates in a power trace. In addition, it allows us to determine if a specific input value is used by a bytecode or not. In contrast to correlation with input values, the negative correlation is not relevant when using it to recognise templates. In this paper, a correlation of 1 is represented as 100% and 0 is represented as 0%. Detailed information about the correlation function is given in Appendix B.

Averaging. As depicted in Figure 2, a single power trace is noisy. Taking the arithmetic mean of a set of traces is a simple but effective technique to remove noise. Figure 3 depicts the average of 10000 power traces of the same Java Card applet using the same input data. Note that, in contrast to Figure 2, a repeated pattern is clearly visible. The techniques described in the rest of this section assume averaged trace sets, as single traces are too noisy.

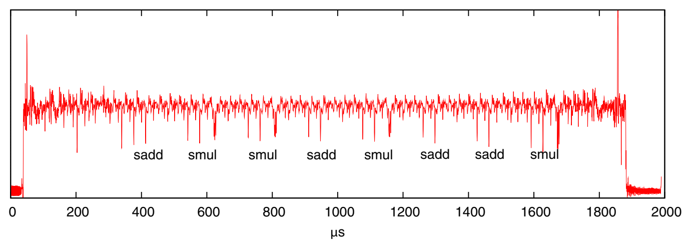


Fig. 3. Average of 10000 power traces. Note that only the interesting region (1 ms to 3 ms in Figure 2) is acquired.

Template determination. In order to recognise bytecodes in a power trace, each bytecode needs to be represented by a unique template. To determine a template for a specific bytecode, a test applet that contains this bytecode is used. For example, the source code fragment depicted in Figure 1 can be used to determine templates of 10 different bytecodes (i.e. `aload`, `baload`, `return`, `s2b`, `sadd`, `sconst_5`, `sload`, `sload_2`, `smul` and `sstore`). Storing frequently occurring bytecode sequences as a single template is also considered. These templates are referred to as *combined templates.*

The execution of the fetch, decode and execute sequence of the JCVM also corresponds to a specific template (referred to as *JCVM template*). This is advantageous, because this template can be utilised to split the power trace into separate parts representing the individual bytecodes. By comparing them with the bytecode of the known Java Card applet, it is possible to store them as the template for that specific bytecode. The same technique can also be used to determine templates of native methods, e.g. a DES operation [7]. It is important to note that the templates considered here are valid only for the specific smart card type used.

Template Recognition. The templates determined in the previous section can be used to process an unknown applet. We developed a program that automatically matches n templates against an average power trace by using the correlation technique described earlier. The result of this program is a set of n traces containing the correlation of the power trace with each template. One example is depicted in Figure 4 where the power trace (shown in red on the first row) and its correlation with templates for `sload`, `baload`, `sadd+s2b+sstore` and `smul+s2b+sstore` respectively are shown. From Figure 4 can be concluded that the bytecode sequence `smul+s2b+sstore` is probably executed three times during the applet execution.

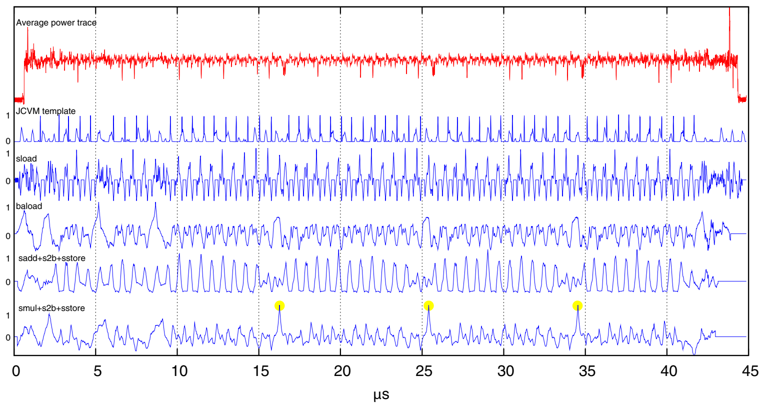


Fig. 4. Result of the template matching process

3 Experimental Results

As our first experiment, we developed a Java Card applet that performs only two addition statements. Table 1 shows the results of the template recognition process as described in Section 2. The first column contains the actual bytecodes that were executed. The second column contains the bytecode with the best correlation, while the third column contains alternative bytecode candidates that have a correlation greater than a predefined threshold (i.e. 50%). The JCVM template is used to cluster the execution trace. Note that the results contain uncertainties and even one error (i.e. on the sixth row, the `aload` bytecode has a better correlation than the actual `sload` bytecode).

Table 1. Example execution trace obtained from the power analysis

Expected	Recognised	Alternatives
sload	sload (93%)	aload (89%)
	JCVM	
sload	sload (92%)	aload (91%), sconst & sstore (57%)
	JCVM	
sadd	sadd (91%)	sload (55%), aload (51%)
	JCVM	
s2b & sstore	s2b & sstore (91%)	sload (51%)
	JCVM	
sload	sload (92%)	aload (78%), sconst & sstore (54%)
	JCVM	
sload	~~aload (92%)~~	sload (91%)
	JCVM	
sadd	sadd (90%)	sload (54%), aload (53%)
	JCVM	
s2b & sstore	s2b & sstore (90%)	sload (53%)

Our second experiment was to attempt distinguishing bytecodes that perform similar operations. Some bytecodes that are available in the JCVM are used to optimise common operations. For example, loading a short value from local variable 2 or 3 can be performed using sload_2 or sload_3 respectively. We performed this measurement at 200 MHz, because distinguishing between similar bytecodes, such as sload_2 and sload_3, is difficult using resampled traces.

We performed 12500 measurements of the power consumption during the execution of an sload_2 bytecode and another 12500 measurements during the execution of an sload_3 bytecode. Figure 5 depicts the difference between sload_2 and sload_3. There is some difference only during a small period of time (i.e. approximately 400ns). Although our experiment indicated the possibility to determine the exact type of sload operation, a lot of traces must be collected making this process very time consuming.

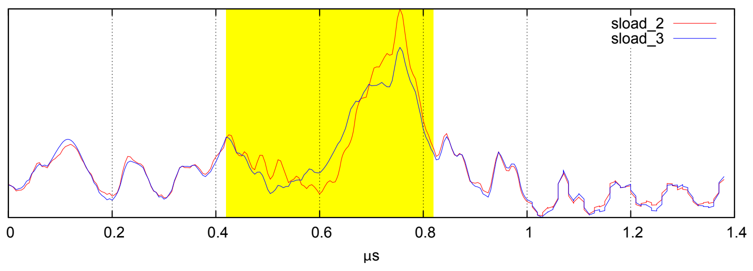


Fig. 5. Difference between sload_2 and sload_3

3.1 Methodology Refinements

Our first experiments indicated that power analysis only, sometimes can not provide enough information to recognise the correct bytecode template. We refined our methodology by identifying additional information sources as will be described in this section.

Impossible Bytecode Sequences. Not all bytecode can follow each other. During the reverse engineering process it is advantageous to keep an operand type stack. Although storing the operands themselves is difficult, storing their types is much easier. Based on the elements on top of the operand type stack, some bytecodes can be excluded from the set of possible follow-up bytecodes. Note that this approach will greatly reduce the search space.

When this technique is applied to the example of Table 1, the impossible bytecode sequence in this example: `sload`, `aload`, `sadd` can be recognised. Because an `sadd` bytecode expects a `short` on top of the operand stack, while an `aload` bytecode pushes an *objectref*, the `aload` must be replaced by an alternative bytecode (i.e. the `sload` bytecode that matches for 91%). This results in: `sload`, `sload` (first alternative), `sadd`. In this case, it is assumed that the `sload` bytecode on line 5 and the `sadd` bytecode on line 7 are correctly recognised.

Unlikely Bytecode Sequences. Besides impossible bytecode sequences, as described above, there are also bytecode sequences that are unlikely to occur even they are allowed by the JCVM. For example, `sconst_0` , `sdiv` (divide by constant 0) is obviously not likely to occur in a normal trace.

Bytecode Statistics. Statistical information about already processed Java Card applets can also be used. Because the bytecode of Java Card applets on a smart card is usually generated by the Java compiler, certain patterns will occur more often than others. For example, experiments reveal that a `for` loop, will always be generated as depicted in Figure 6. Other examples are `i=0` and `i++` which are depicted on lines 1-2 and 5-9 respectively. Saving a template for each of these frequently occurring patterns is advantageous. Experiments showed that templates which contain more samples usually have a less noisy correlation, as depicted in the last trace of Figure 4.

Input Data. Besides correlating a power trace with templates, correlation with input data contained in the command can also be used to determine which bytecode uses input data. The example in Figure 7 depicts the average power trace of the `smul` bytecode. In addition, it also depicts the correlation with the first operand of the `smul` bytecode and the correlation with a random byte, which is not used by the `smul` bytecode. From Figure 7 one can conclude that it is possible to determine if a specific input value is used by a bytecode.

```
      sconst_0
      sstore_2
      goto L2
L1:   // Loop body is inserted here
      sload_2
      sconst_1
      sadd
      s2b
      sstore_2
L2:   sload_2
      bspush    3
      if_scmplt L1
```

Fig. 6. A `for` loop as generated by the Java compiler

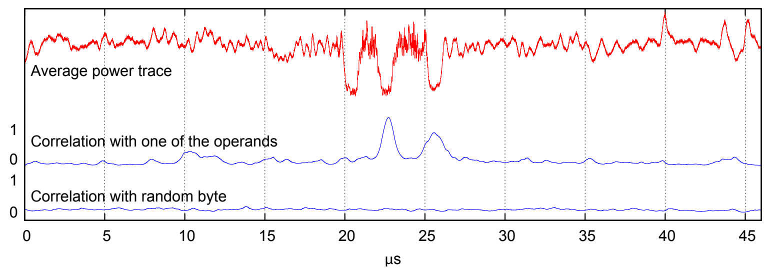


Fig. 7. Correlation between input data and the power trace

Bytecode duration. In some situations the duration of a bytecode execution gives useful information. We found that the duration of a conditional branch bytecode indicates if a branch is taken or not. For example, the duration of the non-taken `if_scmplt` bytecode is approximately 5.75µs. In case the branch is taken the duration increases by 4.5µs to 10.25µs.

Loop Rerolling. Using the techniques described earlier, it is possible to obtain an applet execution trace. In order to reverse engineer a Java Card applet completely, the execution trace should be transformed into structured bytecode. This step is certainly not trivial, because an execution trace is very likely to contain loops.

In the ideal case, the reverse engineering process would generate an execution trace as depicted in Figure 8. This figure shows the execution of a loop which is iterated 3 times.The execution trace can be divided into several parts. First of all, lines 3-6 indicate the presence of a loop. The `goto` statement is used to branch to the conditional part of the loop which loads a `short` value (`sload`), pushes a constant (`bspush`) and branches if the `short` comparison succeeds (`if_scmplt`). Second, the lines following the `goto` statement (i.e. lines 4-6) can be used to split the execution trace of the loop into repetitive parts. The end of the loop is reached when the conditional branch bytecode is not followed by the loop body.

```
1  sconst_0
2  sstore_2
3  goto
4  sload_2
5  bspush     3
6  if_scmplt
7  // Loop body
8  sload_2
9  sconst_1
10 sadd
11 s2b
12 sstore_2
13 sload_2
14 bspush     3
15 if_scmplt
16 // Loop body
17 sload_2
18 sconst_1
19 sadd
20 s2b
21 sstore_2
22 sload_2
23 bspush     3
24 if_scmplt
25 // Loop body
26 sload_2
27 sconst_1
28 sadd
29 s2b
30 sstore_2
31 sload_2
32 bspush     3
33 if_scmplt
```

Fig. 8. Execution trace of the program depicted in Figure 6

In addition, the duration of the conditional branch bytecode may also indicate the end of the loop, as explained earlier.

Besides reconstructing the loop, rerolling the loop has other advantages. First of all it is possible to derive the labels originally used on lines 3, 6, 15, 24 and 33. Second, it is very common that the same loop variable is used in the initialisation, condition and increment part of the loop. Therefore it is likely that the bytecodes on lines 2, 4, 8, 12, 13, 17, 21, 22, 26, 30 and 31 share the same local variable index.

Although this technique works fine for this relatively simple example, it is rather difficult to automate this process. Detecting a nested loop as such is not very difficult, because the nested loop will cause an additional `goto` statement. However, reconstruction of a nested loop is difficult because the conditional part may contain similar statements and the execution traces may contain errors. Moreover the loop may contain conditional statements.

Conditional Branches. Conditional branch bytecodes, such as `if_scmplt`, make the reverse engineering process more difficult. By varying the input data, it is possible that another part of the source code is executed. Without knowledge of the source code it can be difficult to determine on what input data a conditional branch bytecode is dependent. There are two ways to determine such dependency:

– Use correlation between random input data and the power profile of the conditional branch bytecode;
– Inspect the reverse engineered applet first and try to derive what input data is used in the condition.

It is however possible that a varying input data does not affect the conditional branch, for example when it is based on an internal state or data from a random generator. In this case the condition has to be determined from the partially reverse engineered source code.

3.2 Execution Trace Decompilation

When the structured bytecode is available, it is relatively easy to reconstruct source-level expressions. In [11], a technique to automatically decompile Java bytecodes into Java source code is presented. Although the referred paper focuses on decompiling standard Java bytecode, we successfully implemented a Java Card version. Implementation details of this process are outside the scope of this paper.

4 Conclusion and Future Work

In this paper we showed that power analysis can be used to acquire information about executed bytecodes on a Java Card smart card. Using the right equipment and a methodology to determine and recognise bytecode templates, we were able to generate an execution trace of a Java Card applet. Although the tested smart card activates a countermeasure against power analysis when the JCVM is active, we found that this countermeasure is not very effective. Next, we showed that besides power analysis, additional information sources can be used to reduce the number of errors and uncertainties in the execution trace based on the fact that:

– some bytecode sequences cannot occur in a valid Java Card applet;
– some bytecode sequences are very unlikely to occur, although they are valid;
– statistics of other Java Card applets can identify frequently occurring bytecode sequences.
– correlation with input data can be used to determine which variables depend on input data;
– the duration of some bytecodes can provide information. The duration of a conditional branch bytecode indicates if a branch is taken or not.

In addition, we presented techniques to generate structured bytecode from the execution trace using loop rerolling. Most of the time however, this step will be difficult, as the execution trace may also contain nested loops and other conditional statements. On the other hand it may still be possible though, to reverse engineer those parts manually.

Finally we showed that structured bytecode, once it is available, can be decompiled relatively easy to Java source code using algorithms which are also used to decompile regular Java applications.

4.1 Directions for Future Work

There are a couple of topics that will be addressed in the future. First of all, it would be interesting to see if the techniques described in this paper can also be applied to RFIDs (contactless smart cards). Second, we intend to investigate different countermeasures aimed to prevent Java Card applets from being reverse engineered using power analysis. Finally, a program that performs the template determination for all bytecodes automatically would be interesting.

References

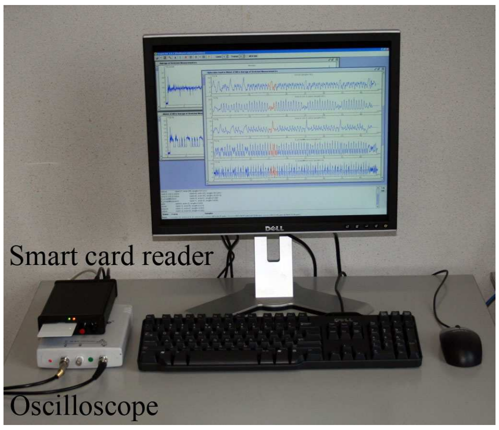
1. Sun Microsystems, Inc. `http://www.sun.com/smi/Press/sunflash/2004-11/sunflash.20041102.1.xml` (2004)
2. Sun Microsystems, Inc. `http://www.sun.com/smi/Press/sunflash/2005-11/` sunflash.20051115.2.xml (2005)
3. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In Wiener, M.J., ed.: CRYPTO. Volume 1666 of Lecture Notes in Computer Science., Springer (1999) 388–397
4. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Koblitz, N., ed.: CRYPTO. Volume 1109 of Lecture Notes in Computer Science., Springer (1996) 104–113
5. Messerges, T.S., Dabbish, E.A., Sloan, R.H.: Examining smart-card security under the threat of power analysis attacks. IEEE Trans. Computers **51**(5) (2002) 541–552
6. Witteman, M.: Advances in smartcard security. Information Security Bulletin **7** (2002) 11–22 Also available at `http://www.riscure.com/articles/ISB0707MW.pdf`.
7. Vermoen, D.: Reverse engineering of java card applets using power analysis (2006) Available at `http://ce.et.tudelft.nl/publicationfiles/1162_634_thesis_Dennis.pdf`.
8. Chen, Z.: Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)
9. Witteman, M.: Java card security. Information Security Bulletin **8** (2003) 291–298 Also available at `http://www.riscure.com/articles/ISB0808MW.pdf`.
10. Press, W., Teukolsky, S., Vettering, W., Flannery, B.: Numerical Recipes in C++ – Second Edition. Cambridge University Press, Cambridge, UK (2002)
11. Proebsting, T.A., Watterson, S.A.: Krakatoa: Decompilation in java (does bytecode reveal source?). In: COOTS, USENIX (1997) 185–198

A Acquisition Framework

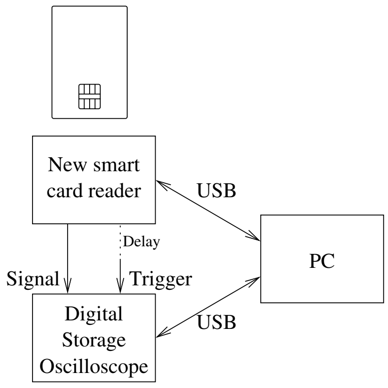
In order to collect power traces, we developed an acquisition system. As depicted in Figure 9, the system consists of a smart card reader, a Digital Storage Oscilloscope (DSO) and a PC. Both the smart card reader and the DSO are connected to the PC using separate USB channels.

Our initial system triggered the oscilloscope using software. Unfortunately this caused the oscilloscope to be triggered at different positions in the applet execution. The time required to execute a Java Card applet, is typically longer than an oscilloscope can store in its memory. Therefore we developed a new smart card reader that automatically triggers the oscilloscope after sending the last byte of a command APDU. The trigger signal can eventually be delayed with µs precision to inspect different parts of applet under test.

The experiments performed in this paper are performed using a 200 MHz DSO that can store approximately one million samples.



(a)



(b)

Fig. 9. The acquisition system

B Correlation Formulas

In order to understand how the correlation between two variables can be computed, some other functions must be defined first. The variance of x is defined as:

$$var(x) = \frac{(\sum x_i - \overline{x})^2}{n-1} \tag{1}$$

where x_i represents the i-th element of x, $\overline{x}$ is the algebraic mean of x, and n is the size of x. The covariance of x and y provides a measure of how much x and y are related and is defined as:

$$cov(x,y) = \frac{\sum (x_i - \overline{x})(y_i - \overline{y})}{n-1} \tag{2}$$

The covariance is difficult to interpret though, because it depends on the scale of the input values. A better measure, independent on the absolute values of the input is given by the correlation function which is defined as:

$$corr(x,y) = \frac{cov(x,y)}{\sqrt{var(x) \cdot var(y)}} \tag{3}$$