POWER ANALYSIS ON SMARTCARD ALGORITHMS USING SIMULATION

Gijs Hollestelle
Wouter Burgers
Jerry den Hartog

Eindhoven, University of Technology

May 2004

Abstract

This paper presents the results from a power analysis of the AES and RSA algorithms by simulation using the PINPAS tool. The PINPAS tool is capable of simulating the power consumption of assembler programs implemented in, amongst others, Hitachi H8/300 assembler. The Hitachi H8/300 is a popular CPU for smartcards. Using the PINPAS tool, the vulnerability for power analysis attacks of straightforward AES and RSA implementations is examined. In case a vulnerability is found countermeasures are added to the implementation that attempt to counter power analysis attacks. After these modifications the analysis is performed again and the new results are compared to the original results.

1 Introduction

In modern times, smartcards are a widely occurring phenomenon. They are used for electronic money, security, the subway, digital signatures and in numerous other places.

Because of the wide use of the smartcard, they also become very interesting for attackers. It is therefore of importance that a smartcard itself is well secured. Up until recently the use of safe and secure encryption algorithms was enough to accomplish this.

The introduction of a new type of attack changed all that. Power analysis attacks use the power signals produced by the smartcard to retrieve information from it that was considered secure. No longer is a mathematical secure smartcard enough. The information leaked by the hardware of the smartcard must also be minimal and of no use to an attacker. This can be accomplished by modifying the hardware itself or by modifying the implementation of the algorithm to include various forms of countermeasures against power analysis attacks.

This paper reports the results we obtained by using the tool created in the PINPAS, or Program INferred Power Analysis in Software, project to analyze the security of the implementation of two common encryption algorithms for smartcards. These are the AES and RSA algorithms.

First the theory behind power analysis is presented along with an introduction to the PINPAS tool. Next we will discuss the AES algorithm. This will include the theory behind the algorithm, our implementation of the algorithm and the results we obtained from the power analysis. After this a similar chapter is devoted to RSA.

This report is based a student research project performed by Gijs Hollestelle and Wouter Burgers, supervised by Jerry den Hartog, within the ECSS group at Eindhoven, University of Technology in the period of September 2003 to December 2003.

2 Power Analysis

2.1 General

When cryptographic algorithms are designed and analyzed, a lot of effort is put into securing them against all forms of mathematical attacks. In practice it is often a lot easier to attack the algorithm by using side-channel information generated by an implementation of the algorithm on certain hardware. In this paper we will be concentrating on smartcard hardware. Sources of side-channel information when running algorithms on smartcards are: power consumption, timing information, electromagnetic emissions and introducing faults into the hardware. In our research we will focus on power consumption.

It is not complicated to measure the power consumption of a smartcard; because the smartcard has no power source of itself it has to rely on an external power source that can be easily monitored, for example, by using a digital oscilloscope as described in [Mess99]. Power analysis uses the fact that the power consumption of a smartcard is based on the hamming weight of the value it is currently processing.[1]

2.2 Simple Power Analysis

Simple Power Analysis or SPA is power analysis based on a single power trace generated by an algorithm run on a smartcard. By observing the trace one can, for example, gain information about the hamming weight of a certain byte of the key when the location in the power trace where it is being used is known. One can also use visible characteristics of the power trace to gain information on the key, for example the visual attack on RSA as described in chapter 5. Here the power trace reveals the order of the multiplication and squaring operations that are being performed in RSA, thereby revealing information on the key.

2.3 Differential Power Analysis

Differential Power Analysis or DPA was discovered by Kocher, see [Koch98]. DPA is a technique where power traces are combined in a statistical manner to obtain information about the algorithm running on the smartcard. DPA works as follows:

- Generate a (large) number of power traces
- Establish an attack point, as described below
- For all key guesses *k* run a function D_k (called a condition) that gives a value 0 or 1, to divide the traces in two sets S_0 and S_1 for each trace.
- For all positions *i* in the power trace, calculate the average power usage (A_{i0} and A_{i1}) for both sets at this position.
- Calculate the difference E_i between (A_{i0} and A_{i1}) for all positions i.

Attacks are possible at points, where the outcome of a calculation can be predicted, by using known information (for example input and output) and information for which all possibilities can be checked (for example a small part of the key). D is a function that calculates the predicted value at the attack point and divides the traces according to this output.

[1] Different smartcard hardware will give different relations between powerconsumption and the data being processed. Here we will assume that the power consumption is directly related to the hamming weight of data being processed.

DPA is based on the fact that E_i will be small at positions *i*, where the power usage is independent of the outcome of the function D (i.e. places where the key is not being used or where the key was guessed incorrectly). At positions *i* where the power usage is dependant on D, E_i will be bigger. This technique can be used to identify the location in the power trace where the key is used and to find the key that has been used.

3 PINPAS

The 'Program INferred Power Analysis in Software' (PINPAS) project, is a project of the TU/e in cooperation with TNO-TPD.

The project's goal is to develop an integrated support environment for countering side-channel attacks (focusing on power analysis), selecting the most suitable type of smartcard and to determine the sensitivity of an implementation to power analysis. By describing an implementation of a cryptographic algorithm in a suitable programming language with dedicated data types, constructions to counter power analysis and by compiling it into generic assembly code the project has the following advantages:

- Power analysis can be performed at the software level.
- Attack scenarios based on the algorithm can be developed and automatically executed

The PINPAS tool, created in the project, has been designed by J. den Hartog et al and is described in [HVVVW03]. This tool allows for power analysis of software that is not yet implemented on a real smartcard. It can be used in the design process to make an implementation more secure against power attacks. The PINPAS tool can simulate smartcard hardware and run a program written in assembler code for the actual smartcard hardware. It calculates a power trace based on this simulation. The tool can also perform DPA and SPA analysis on the generated traces, or export these traces to allow another program to perform the analysis.

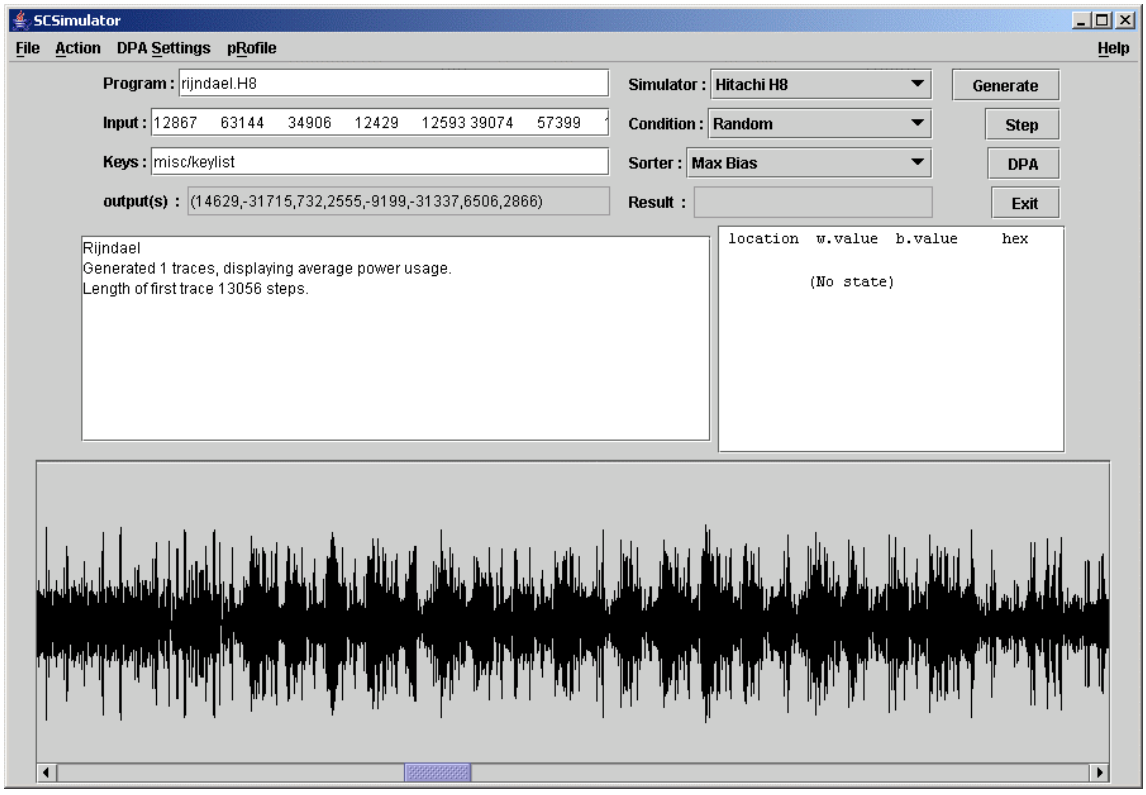

Figure 1: The PINPAS tool showing a power trace of the AES algorithm.

4 AES

In this chapter the results obtained from analyzing the Advanced Encryption Standard (AES) algorithm are presented. This includes the workings of the algorithm, implementation and power analysis results. We have limited ourselves to using a cipher key of 128 bits and a block size of 128 bits, even though the algorithm also supports different key and block sizes. It is expected that analysis using other key and block sizes would give similar results.

4.1 The Algorithm

4.1.1 Global

The current AES is the cipher Rijndael with the added constraint that the key size is limited to 128, 192 or 256 bits and the block size is limited to 128 bits. Rijndael was developed by Joan Daemen and Vincent Rijmen and is fully documented in [DaRij99].

In many ways, AES is a simple cipher. It takes an input of 128 bits (which equals 16 bytes) and orders this as a 4x4 matrix. The key, in our environment always 128 bits, is ordered in a similar fashion.

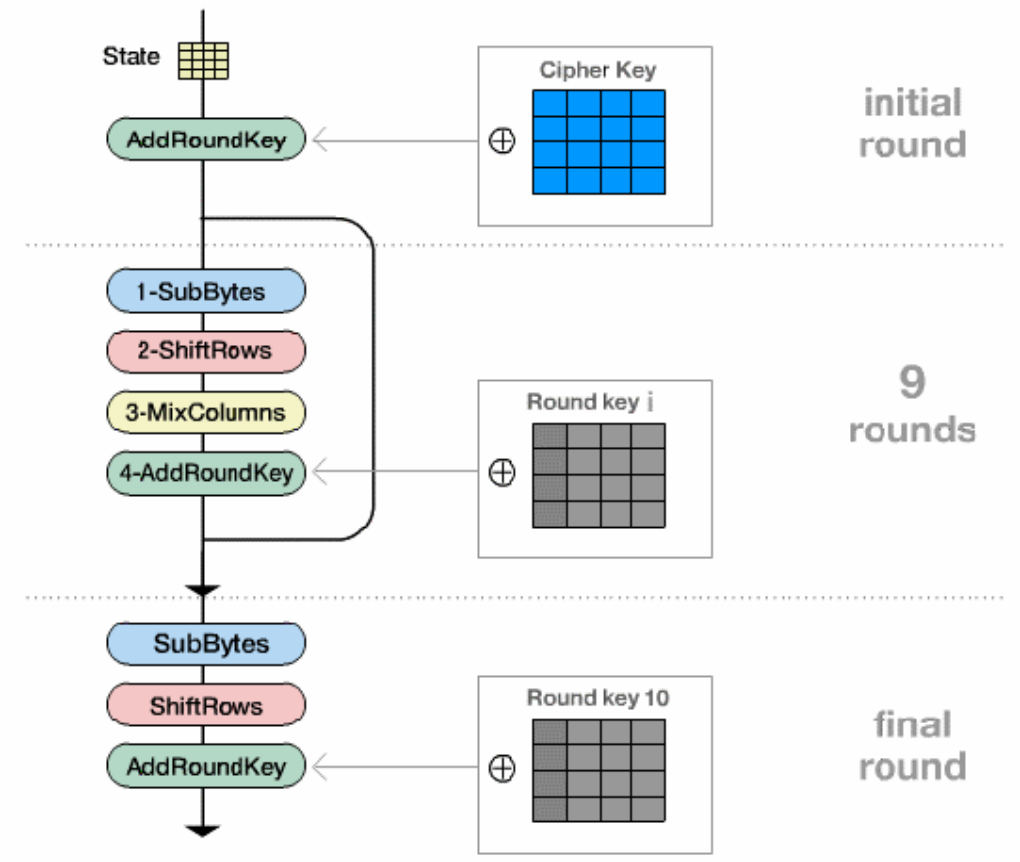

Figure 2: A global overview of the AES algorithm. Image taken from [RijnAnim]

The algorithm consists of a single AddRoundKey, using the cipher key, followed by 9 regular rounds each consisting of 4 steps and a final round. The steps of the regular rounds are SubBytes, ShiftRows, MixColumn and AddRoundKey. The final round skips the MixColumn

step. Every round requires its own round key. These keys can be generated in advance or one per round by adding an extra round step.

We will now discuss what calculations are performed in each of the 4 steps mentioned.

4.1.2 SubBytes

In the SubBytes step, every byte from the current state block is replaced by the matching byte from the S-Box. The S-Box is a simple invertible lookup table consisting of 256 values. The specification of AES [DaRij99] contains an explanation about how the S-Box can be calculated. We will not discuss that any further here.

4.1.3 ShiftRows

In the ShiftRows step, the bytes from the current state block, still ordered as the 4x4 column major matrix are shifted as follows:

```
From:               To:
01 05 09 13         01 05 09 13
02 06 10 14         06 10 14 02
03 07 11 15         11 15 03 07
04 08 12 16         16 04 08 12
```

In our situation, where we limit ourselves to 128bit keys and 128bit input blocks, this equals shifting row i, i places left. Note that for other key / block size combinations the algorithm using different shiftings that may not have this property.

4.1.4 MixColumn

In the MixColumn step, the columns of the current state block are considered as polynomials over GF(2^8) and multiplied modulo the polynomial x^4 + 1 with a fixed polynomial given by: ‘3’ x^3 + ‘1’ x^2 + ‘1’ x + ‘2’. This can be written as multiplying each column with the following matrix:

```
2 3 1 1
1 2 3 1
1 1 2 3
3 1 1 2
```

4.1.5 AddRoundKey

The last step of a regular round is AddRoundKey. In this round the algorithm performs a simple XOR on the current state block and the current round key.

4.1.5.1 Key Schedule

In the Key Schedule process all the round keys that are needed are calculated. The initial setup step of the algorithm uses the cipher key. The 10 regular rounds each need their own round key.

The key for round *i* is generated by performing a series of calculations on the previous key. This includes several XOR operations using the previous key or a round constant, S-Box lookups and permutations. See [DaRij99] for a detailed description.
The round keys can all be generated in advance and stored in a buffer large enough to hold all 10 round keys and the cipher key, or the key for round *i* can be generated in round *i*

somewhere before the AddRoundKey step (which is the only step that uses the round key). The latter approach only requires a buffer large enough to hold a single key.

4.1.6 Encryption and Decryption

Encryption is done with the algorithm described above. The decryption algorithm is the inverse of the encipher algorithm.

This means the order of the round transformations are reversed and the transformations themselves are replaced by their inverse transformations. Note that the inverse of AddRoundKey is AddRoundKey itself. This means the reverse of a round is given by:

```
AddRoundKey( State, RoundKey );
InvMixColumn( State );
InvShiftRow( State );
InvSubByte( State );
```

The decryption algorithm omits the InvMixColumn step in the initial round and performs an AddRoundKey using the cipher key after completing all regular rounds.

4.2 Implementation

4.2.1 Target Platform

For use in the PINPAS tool an implementation of the algorithm for the Hitachi H8/300 processor needed to be created. The assembler version of the algorithm was created by compiling a C implementation of the algorithm.

4.2.2 Implementation Goals

Before implementation of the algorithm in C was started a number of implementation goals were set. A readable and simple version of the algorithm was preferred, but not at all costs. The resulting compiled assembler version needed to be efficient enough for use in the PINPAS tool.

All round constants and the S-Box table are stored entirely in memory during program execution and are thus not generated on the fly. To not put any further pressure on the memory usage (the memory size is usually very limited on smart cards) only a single round key at a time is stored and a new round key is calculated every round, overwriting the previous round key. Because if all RoundKeys were stored on the card, they would all have to be stored in secured memory locations. Because the round constants and S-Box table are publicly known, they can be stored in less secure ROM memory and they do not have to be recalculated for every card as these are constants for the AES algorithm.

4.2.3 C Implementation

Implementing the AES algorithm is pretty straightforward. The resulting implementation can be found in appendix A.

4.2.4 Optimizing the C Implementation

Compiling the initial implementation of the AES algorithm to H8/300 assembler, results in an unnecessary inefficient program. The number of steps performed is just too high. Because a large number of power traces need to be generated by PINPAS for the Differential Power

Analysis an inefficient program can quickly lengthen the calculation time to unacceptable proportions. An inefficient program is also not realistic for smartcards.

Therefore a number of steps are taken to optimize the C version in such a way that the resulting assembler program is usable.

These steps include, but are not limited to, removing all local variables (thus accessing only global memory), rewriting all functions not to include parameters, rewriting all functions not to include return values and replacing all loops (`for`, `while`) by their equivalent code that only uses labels and `goto` statements. Finally we profiled the version to see which functions took the most steps and actions were taken to optimize these functions (for example, by writing out an entire loop). The final C implementation can be found in appendix B.

4.2.5 Final Assembler Implementation

All steps taken to optimize the C implementation significantly reduce the number of steps the assembler version takes. Our final assembler version has a constant running time, except for the MixColumn step, and takes approximately 13500 steps. This proved to be more than efficient enough for use in the PINPAS tool. All our attack points lie before the first regular round (which includes the first MixColumn step), so the part of the program that we evaluate in the PINPAS tool has a constant running time.

The relevant code of the assembler version can be found in appendix C.

4.3 Differential Power Analysis

4.3.1 Per-Byte Attack

4.3.1.1 Setup

4.3.1.1.1 Attack point

The first step is to identify a place in which the algorithm can be easily attacked. The goal of our attack is to retrieve the cipher key. In AES an obvious place for such an attack is in the initial AddRoundKey, which takes place before the first regular round. In this step the input state block is XOR-ed with the cipher key (which we are looking for).

The C implementation is given below (The H8/300 assembler version is a direct descendant of this version):

```
01:   void AddRoundKey()
02:   {
03:     for (i = 0; i < 16; i++)
04:     {
05:        inputdata[ i ] = inputdata[ i ] ^ key[ i ];
06:      }
07:   }
```

Note line number 5. This is the line in which our DPA attack is possible. The right hand side of the expression on that line equals a single assembler instruction in which a byte of the input data is XOR-ed with a byte of the cipher key. Since the values of the input are known and there are only 256 possibilities for the value of that particular byte of the key a DPA attack is possible.

Attacking at this point was made slightly more convenient by modifying the assembler version as follows. First the execution of the algorithm is halted after the first AddRoundKey. We can do this, since we're attacking before this point and everything that happens after this point is of no interest to us nor relevant to the attack. We've also configured PINPAS to only record power signals concerning to the XOR operation on the input data and cipher key (line number 5 of the C implementation).

This limits the number of steps that PINPAS has to calculate for a single execution of the program severely and results in a power trace length of 16 (since there are 16 bytes in the cipher key). If an actual physical DPA is performed one also only needs to inspect those 16 values.

Note that the AddRoundKey is not the only place where such an attack is possible, also the SubBytes round is vulnerable to a similar attack. When securing Rijndael against DPA one has to consider the other rounds as well.

4.3.1.1.2 Condition

For an attack using PINPAS a DPA condition is required, which we've called the Rijndael Condition. This condition is given below (only relevant code is shown):

```
01:     public boolean select( pinpas.TraceInterface t )
02:          throws UnsortableException
03:     {
04:       int attackedbyte;
05:
06:       pinpas.Word[] inputs = t.getInputs();
07:
08:       if ( mode % 2 == 1 )
09:         attackedbyte = inputs[ (mode-1)/2 ].getByte(0);
10:       else
11:         attackedbyte = inputs[ (mode)/2 ].getByte(1);
12:
13:       // XOR the byte with the key.
14:       attackedbyte ^= key;
15:
16:       // Calculate the hamming weight.
17:       int wgt = pinpas.Util.hammingWgt( attackedbyte, 8 );
18:
19:       // Return true or false depending on the hamming weight.
20:       if ( wgt > 4 ) return true;
21:       if ( wgt < 4 ) return false;
22:
23:       throw new UnsortableException();
24:     }
```

This condition will divide traces into two groups, by calculating the predicted hamming weight for the result of the XOR operation used in the AddRoundKey step. It does this by using the known input and trying all 256 possible values for the current byte of the key.

4.3.1.1.3 Input

The test is performed three times, each time with different input sets of different sizes. PINPAS was fed with 400, 2000 and 4000 random input values. This results in 50, 250 and 500 power traces respectively.

4.3.1.2 Initial Results

The table below shows the results of the initial Differential Power Analysis on the straightforward AES implementation:

Byte	DPA Result (50 traces)		DPA Result (250 traces)		DPA Result (500 traces)		Correct value	
0	182	(**1**01**10**1**0**)	59	(001**1**1011)	43	(00101011)	43	(00101011)
1	84	(01**0**1**0**1**0**0)	126	(01111110)	126	(01111110)	126	(01111110)
2	149	(**1**0010101)	21	(00010101)	21	(00010101)	21	(00010101)
3	48	(00**1**10**00**0)	86	(0**1**010110)	22	(00010110)	22	(00010110)
4	146	(**1**0**0**1**0**0**1**0)	40	(00101000)	40	(00101000)	40	(00101000)
5	175	(10101111)	174	(10101110)	174	(10101110)	174	(10101110)
6	199	(110**0**0**1**1)	210	(11010010)	210	(11010010)	210	(11010010)
7	160	(10100**00**0)	166	(10100110)	166	(10100110)	166	(10100110)
8	82	(**0**1**0**1**0**0**1**0)	171	(10101011)	171	(10101011)	171	(10101011)
9	243	(11110011)	241	(1111**00**1)	247	(11110111)	247	(11110111)
10	52	(00**1**1010**0**)	21	(00010101)	21	(00010101)	21	(00010101)
11	136	(10001000)	136	(10001000)	136	(10001000)	136	(10001000)
12	9	(00001001)	9	(00001001)	9	(00001001)	9	(00001001)
13	139	(1**0**001**0**11)	207	(11001111)	207	(11001111)	207	(11001111)
14	236	(**1**1**1**011**00**)	79	(01001111)	79	(00111100)	79	(01001111)
15	56	(00111**0**00)	60	(00111100)	60	(00111100)	60	(00111100)

It is clear from the table that 50 power traces aren't sufficient to correctly retrieve the key. The results with 250 power traces have improved vastly, but three bytes of the key are still guessed incorrect, although the errors are only marginal (1 and 2 bits).

A DPA with 500 traces retrieves all bytes of the key correctly. However, experiments show that this is not necessarily always the case. A different set of 500 traces provided us with a situation in which 15 of the 16 bytes of the key were guessed correctly and 1 byte was guessed wrong, with a 1 bit error.

4.3.1.3 Countermeasures

The – relatively simple – DPA attack performed above is possible because the implementation contains absolutely no countermeasures against such an attack. The program performs the same calculation (using the cipher key) at exactly the same place during each execution of the program.

As a countermeasure against this situation a technique called *randomization* has been used to strenghten the implementation. This technique randomizes the execution order of a series of instructions. This helps to protect against DPA attacks as a DPA attack relies on the same instruction being executed at the same time in each run of the program.

The *randomization* countermeasure was implemented in the AddRoundKey step as follows:

```
01:   void AddRoundKey()
```

```
02:   {
03:     // Create random permutation from 0..15.
04:     for (tmp = 0; tmp < 16; tmp++)
05:     {
06:       // Get two random indices in the range 0..15
07:       i = rand( 16 );
08:       j = rand( 16 );
09:
10:       // Store value at first index.
11:       tmp2 = order[ i ];
12:
13:       // Overwrite value at first index with value from second index.
14:       order[ i ] = order[ j ];
15:
16:       // Write temporary stored value from 1st index to 2nd index.
17:       order[ j ] = order[ i ];
18:     }
19:
20:     // Execute XOR in the above calculated order.
21:     for (i = 0; i < 16; i++)
22:     {
23:        inputdata[ order[ i ] ] ^= key[ order[ i ] ];
24:     }
25:   }
```

As one can see above, this countermeasure is not hard to add to the implementation, yet it should prove efficient enough to make simple DPA attacks, like in section 4.3.1.2, impossible. We will explore that in the next subsection.

Another advantage of this countermeasure is that it doesn't make the execution time of the algorithm significantly larger. It only increases the number of steps required a little and the program maintains its constant runtime.

This technique, however, does require that a random number generator is present on the smart card.

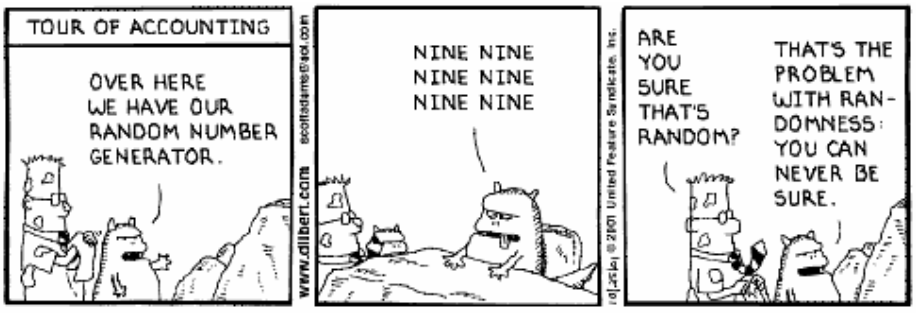

Figure 3: Dilbert on random number generators.

4.3.1.4 Final Results

After implementing the countermeasures a DPA was performed again. This time, off course, we hoped that the correct key would not be found. DPA was performed again with trace sets of three different sizes, as in the previous test. The results are shown in the table below.

Byte	DPA Result	DPA Result	DPA Result	Correct value

	(50 traces)		(250 traces)		(500 traces)			
0	39	(0010**01**11)	2	(00**000**01**0**)	2	(00**000**01**0**)	43	(00101011)
1	41	(0**0**1**0**1**001**)	94	(01**0**11110)	138	(**1000**1**0**10)	126	(01111110)
2	235	(**11101**011)	119	(0**11**101**1**1)	32	(00100**000**)	21	(00010101)
3	239	(**11101**111)	91	(0**1**0**11**0**11**)	6	(0000**0**110)	22	(00010110)
4	178	(**1**01**10**0**1**0)	138	(**10**001**0**10)	21	(00**0101**0**1**)	40	(00101000)
5	28	(**00**0**1**1**0**0)	200	(1**10**01**00**0)	50	(**0**01**100**10)	174	(10101110)
6	70	(**0**10**00**110)	222	(1101**1**110)	97	(**0**1**100**0**1**)	210	(11010010)
7	116	(**01**1**1**01**0**0)	119	(**01**1**1**011**1**)	220	(1**1011**1**0**0)	166	(10100110)
8	205	(1**1**001**1**0**1**)	147	(10**01**011)	14	(**00**001**1**0)	171	(10101011)
9	219	(11**0**1**1**011)	1	(**0000000**1)	140	(1**0001100**)	247	(11110111)
10	58	(00**111010**)	140	(**100**0**1**10**0**)	85	(0**1**010101)	21	(00010101)
11	178	(10**110**0**1**0)	70	(**01**00**11**0)	100	(**01100**100)	136	(10001000)
12	179	(**10110**0**1**1)	37	(00**100**101)	112	(0**1110**000)	9	(00001001)
13	105	(**0**11**0**1**00**1)	105	(**0**11**0**1**00**1)	223	(110**1**1111)	207	(11001111)
14	190	(**10**1**1**111**0**)	2	(0**0000**0**1**0)	67	(0100**00**11)	79	(01001111)
15	93	(0**1**011101)	38	(001**00**11**0**)	140	(**1000**1100)	60	(00111100)

As expected the correct key is not found. Even in the situation with 500 traces, which guessed all bytes of the key correct in the straightforward implementation without any countermeasures, none of the bytes of the key are found. Three bytes are differing by only one bit, one by two bits and two by three bits. All others differ by 4 bits or more.

As a stress test PINPAS was used again on a set of 8000 traces. This is 16 times the amount of traces that was needed to retrieve the correct key for the implementation without countermeasures.

Even in this situation the correct key is not found as can be seen in the table below. It should be noted that the average number of wrong bits is reduced. However, the number of wrong bits is still too significant to make any assumptions or reasonable guesses about the cipher key.

Byte	DPA Result (8000 traces)		Correct value	
0	40	(00101**00**)	43	(00101011)
1	94	(01**0**11110)	126	(01111110)
2	17	(00010**0**01)	21	(00010101)
3	3	(000**00**11)	22	(00010110)
4	32	(00100000)	40	(00101000)
5	130	(10**0000**10)	174	(10101110)
6	234	(11**101**010)	210	(11010010)
7	2	(**00**000**0**10)	166	(10100110)

8	134	(10**00**1**1**0)	171	(10101011)
9	150	(1**00**1011**0**)	247	(11110111)
10	20	(00010100)	21	(00010101)
11	12	(**0**0001100)	136	(10001000)
12	18	(000**10**0**10**)	9	(00001001)
13	183	(1**011**01**1**1)	207	(11001111)
14	5	(**0**000**1**0**1**)	79	(01001111)
15	9	(00**0**1**0**0**1**)	60	(00111100)

4.3.2 Per-Bit Attack

Although the per-byte attack described above already successfully retrieves the key of the AES implementation, we also present a per-bit attack on AES below. This can be considered a proof of concept that per-bit attacks are also possible to simulate using PINPAS. This is useful because certain algorithms or cards might be safe from per-byte attacks, but vulnerable to per-bit attacks.

4.3.2.1 Setup

4.3.2.1.1 Attack Point

The attack point is identical to the one used in the per-byte attack, but the per-bit attack attempts to retrieve the key bit by bit instead of byte by byte.

It does this by inspecting the bit that it is attacking and sorting the entire byte depending on the bits value. If the bits value is 1, the hamming weight of the byte is expected to be high on average and if the bits value is 0, it is expected to be low.

Since we're attacking per-bit, the key list in PINPAS is reduced to {0,1}.

4.3.2.1.2 Condition

The relevant part of the condition file for the per-bit AES attack is given below:

```
01:   public boolean select( pinpas.TraceInterface t )
02:   throws UnsortableException
03:   { int attackedbyte;
04:     pinpas.Word[] inputs = t.getInputs();
05:
06:     // Extract the required byte from the input
07:     int modebit = (mode % 8);
08:     int modebyte = (mode - modebit)/8;
09:
10:     if (modebyte % 2 == 1) {
11:       attackedbyte = inputs[(modebyte-1)/2].getByte(0);
12:     } else {
13:       attackedbyte = inputs[(modebyte)/2].getByte(1);
14:     }
15:
16:     // Return true if the input bit xor keybit != 0
17:     return (((attackedbyte >> modebit) & 1) ^ key != 0);
18:   }
```

4.3.2.1.3 Input

Although the key list is now much smaller, the list of input needs to be much bigger: a per-bit attack requires significantly more traces. We've performed the test three times, once with 4000 inputs, once with 8000 inputs and once with 100000 inputs. This results in 500, 1000 and 12500 traces respectively.

4.3.2.2 Results

Since we're attacking per-bit it's not much use to show a table with the byte-wise results that contains the number of erroneous bits, so we simply present the outcome of the test of whether the key was actually found or not:

Number of traces	Key found
500	No
1000	Yes
12500	Yes

A number of 500 traces was enough to retrieve the key in a per-byte attack, but as expected isn't enough to retrieve the key in a per-bit attack. Since a per-bit DPA attack is much faster (only 2 keys to test) we were able to perform an attack using 12500 traces, which successfully retrieved the key. However, a final test showed that 1000 traces were already enough to discover the key.

4.3.3 Conclusion

When AES is implemented in the way described in [DaRij99], it is very likely that the implementation is vulnerable to a DPA attack, because in the reference implementation the key is XOR-ed directly with the input in the first execution of AddRoundKey. This is almost the ideal case for a DPA attack based on Hamming weights.

This problem can be resolved, for example, by using the randomization countermeasure described in section 4.3.1.3. If this measure is implemented, DPA in AddRoundKey will be a lot more difficult but in theory not impossible, as the randomization should be canceled out if enough samples are used. We have, however, not been able to successfully use the attack above on the improved version using the PINPAS tool.

More advanced countermeasures are described in [Gal03] and could be used to protect the algorithm against other attacks (such as timing attacks in MixColumns).

5 RSA

In this chapter the results obtained from analyzing the RSA algorithm are presented. They are presented in a similar fashion as those in the previous chapter.

5.1 The Algorithm

5.1.1 Global

The RSA algorithm was designed by Rivest, Shamir and Adleman in 1978. It is a public key cryptosystem that can also be used for signing messages. The algorithm makes use of 3 mathematical properties (as described in [RivSA78] and [Til99]):

1. Exponentiation modulo a composite number *n*, i.e. computing *c* from $c = m^e \pmod n$ for given *m* and *e*, is a relatively simple operation.
2. The opposite problem of taking roots modulo a large, composite number *n*, i.e. computing *m* from $c = m^e \pmod n$ for given *c* and *e*, is, in general, believed to be intractable.
3. If the prime factorization of *n* is known, the problem of taking roots modulo *n* is feasible.

Property 1 makes encoding and decoding possible, property 2 ensures that decoding is not possible without the correct key and property 3 is needed in the setup of the algorithm (see below).

5.1.2 Algorithm Setup

In order to use RSA some pre-calculations have to be made. RSA is based on calculating powers of large integers modulo a large composite number. These numbers have to be generated before the system can be used. We want to create a public key *eB* and a corresponding private key dB. These keys are constructed as follows:

- two large primes *pB* and *qB* are chosen
- the modulus *nB* is the product of *pB* and *qB*
- public key *eB* is a randomly chosen number such that GCD(*eB*, (*pB*-1) * (*qB*-1)) = 1
- the private key *dB* is the multiplicative inverse of *eB* modulo (*pB*-1) * (*qB*-1)

The public key *eB* and the modulus *nB* are made public while the rest of the numbers are kept secret. We have used 512 bit numbers for each which is a commonly used length for keys in the RSA algorithm.

5.1.3 Encryption and Decryption

After this setup (which can be done easily using, for example, Mathematica as shown in Appendix D) encryption of plain text message *m* using public key *eB* to cipher text c is accomplished by calculating: $c = m^{eB} \bmod n$.

The message *m* can be recovered from *c* by calculating $m = c^{dB} \bmod n$.

5.2 Implementation

5.2.1 Target Platform

As with the AES implementation, the target platform is again the Hitachi H8/300 processor. This time,however, we assume there is a coprocessor present. For more information about this coprocessor see section 5.2.3.

5.2.2 Implementation Goals

As before, the resulting implementation needs to be efficient enough for use in the PINPAS tool.

To implement an RSA encryption or decryption a program has to be constructed to calculate a^b mod c for given *a, b* and *c*. This is often called a PowerMod and can be implemented quite efficiently when efficient routines exist to multiply and square numbers modulo *n*.

5.2.3 The Coprocessor

As said before, the implementation of RSA for use on smartcards requires efficient routines to perform multiply and square modulo *n* operations. These routines are typically carried out by a coprocessor on the smartcard, which carries a set of mathematical operations that can prove useful in security algorithms.

In our case, we've taken a virtual coprocessor with a minimal amount of instructions: only two, a BIGMUL or BigMultiply instruction and a BIGSQR or BigSquare instruction. This virtual coprocessor is simulated by the PINPAS tool.

The BIGMUL and BIGSQR instructions were implemented in Java for use in the PINPAS tool using Java's `java.math.BigInteger` class.

5.2.4 C Implementation

When the BIGMUL and BIGSQR operations exist the following algorithm (given in pseudo C code) can be used to calculate a^b mod c. This algorithm is also described in [Til99].

```
01:   r = 1;
02:   for (x = 0; x < bitlength( b ); x++)
03:   {
04:     r = BIGSQR( r, n );
05:     if ( bit( b, x ) == 1 )
06:     {
07:        r = BIGMUL( r, a, n );
08:     }
09:   }
```

5.2.5 Final Assembler Implementation

The algorithm above was implemented in H8 assembler by hand.

The code can be found in appendix E.

5.3 Simple Power Analysis

5.3.1 A Timing Attack

5.3.1.1 What are Timing Attacks?

Timing attacks are attacks which use the timing information of algorithms whose control flow depends on the key such as the RSA implementation given in section 5.2. The timing attack that we will be looking is a form of SPA, Simple Power Analysis, which combines timing and power consumption information. In this way on can use the timing of certain sections of the algorithm rather than only the duration of the whole algorithm. See, for example, [Koch96]

for more information. In principle, if an SPA attack is possible, it can be done on a single trace.

The attack is based upon the assumption that you are able to distinguish patterns in the output, in the RSA case for example that you can see the difference between a BIGMUL and a BIGSQR instruction being performed. In the PINPAS tool this can be simulated for example by letting BIGMUL create a different power trail then BIGSQR. Any visible detectable difference between the two will do. When performing timing attacks on real smartcards the differences might be harder to observe.

5.3.1.2 Visualizing a Timing Attack using PINPAS

Below is the visual trace output of an example timing attack performed using the PINPAS tool in which the smaller squares represent the BIGSQR and the bigger squares represent the BIGMUL instructions. The key can now be easily deduced from the output. Because a 0 in the key results in a BIGSQR, and a 1 in a BIGSQR and a BIGMUL we obtain the following result by examining the trace:

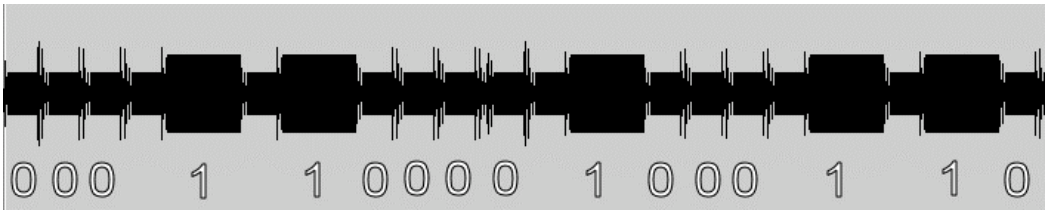

Figure 4: An SPA attack on a power trace of a straightforward RSA implementation.

Which indeed recovers the first 16 bits of the key, the rest of the key can be recovered in the same way.

5.3.1.3 Automating a Timing Attack using PINPAS

To automate the visual attack described above one could place a <POSITION> command in the RSA assembler implementation at the beginning of the BIT loop. The <POSITION> command reports the current position in the power trace.

The table below contains these trace positions and their difference to the previous position. Note that every 8th bit the assembler implementation contains a so called byte loop that will add 5 to the position.

Trace position	Difference	Guessed key bit	Real key bit
163	n/a	n/a	n/a
191	28	0	0
219	28	0	0
247	28	0	0
325	78	1	1
403	78	1	1
431	28	0	0
459	28	0	0
492	28 (+ 5 for byte loop)	0	0
520	28	0	0

598	78	1	1
626	28	0	0
654	28	0	0
682	28	0	0
760	78	1	1
838	78	1	1
871	28 (+ 5 for byte loop)	0	0

5.3.1.4 Protection against Timing Attacks

By introducing a fake BIGMUL when no BIGMUL is to be performed we can make the running time of the program constant and thus independent of the key that was used. The fake BIGMUL performs a multiplication on some bogus data.

Below is an image of the resulting power trace of this version of the algorithm, the attack described in 5.3.2 can no longer be performed.

Figure 5: A power trace of an RSA implementation with fake multiplication.

Also the attack using the <POSITION> command as described in 5.3.3 no longer works:

Trace position	Difference	Guessed key bit	Real key bit
163	n/a	n/a	n/a
242	79	?	0
321	79	?	0
400	79	?	0
479	79	?	1
558	79	?	1
637	79	?	0
716	79	?	0
800	79 (+ 5 for byte loop)	?	0

5.4 Differential Power Analysis

5.4.1 Setup

5.4.1.1 Attack Point

In contrary to AES, an obvious attack point for a DPA attack is not immediately available. This is because all instructions that write to the output data or read from the input data are performed on the coprocessor. The coprocessor is in our case simulated by the PINPAS tool in Java and therefore no (internal) power consumption information is available for that instruction.

Since at the end of program execution the output is known, it was investigated if the key could be found by using DPA to retrieve the key bit by bit starting with the last bit of the key and working our way back to the front.

This required us to predict the value of the intermediate output results. Which are as follows depending on the bit of the key:

	1	0
$\_output_{n-1}$	sqrt($\_output_n$ / input) mod nB	sqrt($\_output_n$) mod nB

However, rule 2 from section 5.1 states that this problem of taking roots modulo a large, composite number nB is, in general, believed to be intractable. Therefore we have to conclude that this type of DPA attack is not possible.

In our simple coprocessor the intermediate data is written back to memory in each step. This allows one to do a DPA attack on these intermediate values. The power consumption of the coprocessor instructions was simulated using the PINPAS tool defaults. A current limitation in these default settings prevents one from seeing the dependency on the complete intermediate result.

As a workaround for this issue we added the following two instructions at the end of the BITLOOP to show the dependency on the intermediate data:

1. mov.w #0,r5
2. mov.b @(_result,r5), r6l;

Note that this issue can also be overcome by creating a custom `power profile' for the instructions of the coprocessor instead of using the defaults.

This moves the first byte of the intermediate result into register six. Since the first byte of the intermediate result depends on the entire input this already gives us enough information to perform a DPA attack.

A more advanced coprocessor may contain an internal buffer where intermediate results are stored. In that case the use of the buffer by the coprocessor should leak minimal information, otherwise any RSA implementation, on smartcards that use this coprocessor, will be vulnerable to a DPA attack. Analysis also shows that a coprocessor with an internal buffer should also support a fake multiplication instruction that can be used to protect RSA implementations, on smartcards that use this coprocessor, against timing attacks as seen in section 5.3.1.

5.4.1.2 Condition

The condition file for the RSA attack is given below:

```
01:   public boolean select( pinpas.TraceInterface t )
02:   throws UnsortableException
03:   { int attackedbyte;
04:     int x;
05:     byte[] inputbytes = new byte[64];
06:     pinpas.Word[] inputs = t.getInputs();
07:
```

```
08:     // Read input
09:     for (x=0;x<32;x++) {
10:       inputbytes [x*2+1] = (byte) (inputs[x].getByte(0));
11:       inputbytes [x*2] = (byte) (inputs[x].getByte(1));
12:     }
13:
14:     // Transfer input data to a biginteger
15:     BigInteger inputdata = new BigInteger(inputbytes);
16:
17:     // Set the modulus and the key
18:     BigInteger modulus = new BigInteger(// Input modulus here);
19:     BigInteger bigkey = BigInteger.valueOf(key).add(prevkey);
20:
21:     // Calculate the first byte of the output
22:     attackedbyte = inputdata.modPow(bigkey,modulus).toByteArray()[0];
23:
24:     int wgt = pinpas.Util.hammingWgt( attackedbyte, 8 );
25:     if ( wgt > 4 ) return true;
26:     if ( wgt < 4 ) return false;
27:     throw new UnsortableException();
28:   }
```

This condition divides traces into two groups, by calculating the predicted hamming weight for the first byte of the result, by calculating an exponentiation of the known input to the power of (prevkey + guessed key). Here prevkey is the part of the key that has already been recovered and for guessed key all 256 possible values for this byte will be tried.

5.4.1.3 Input

PINPAS was fed with 2000 and 4000 random input values. This results in 62 and 125 power traces respectively.

5.4.2 Initial Results

The table below contains the results for the first 32 bytes of the key. Note that when the tool attempts to attack byte *i*, it uses key information from all bytes up to byte *i*. This means that in a real attack all bytes up to byte *i* must have been guessed correctly before byte *i* itself can be retrieved using DPA. When a mistake is made in one of the previous key bytes the peaks that can be observed in the graphical display of the PINPAS tool will be much smaller, indicating that one of the previous key bytes was guessed incorrectly, the attacker can use this information by going back to a previous key value. Note that in the table below we manually corrected the invalid key values, so that these errors would not influence the rest of the results.

Byte	DPA Result (62 traces)	DPA Result (125 traces)	Correct Result
0	24	24	24
1	70	70	70
2	242	242	242
3	**70**	148	148
4	**69**	37	37
5	72	72	72
6	**159**	127	127
7	**185**	119	119
8	**18**	19	19
9	**229**	115	115

10	113	113	113
11	**36**	102	102
12	130	130	130
13	29	29	29
14	98	98	98
15	7	7	7
16	243	243	243
17	168	168	168
18	174	174	174
19	146	146	146
20	149	149	149
21	6	6	6
22	**132**	86	86
23	221	221	221
24	**37**	151	151
25	111	111	111
26	141	141	141
27	214	214	214
28	103	103	103
29	31	31	31
30	**9**	119	119
31	**255**	248	248

In the special case that the first byte of the key is 0, the intermediate result will be the same for any input and there will be no peak in PINPAS at all, if this is the case one can perform the attack, starting at the next byte of the key until peaks begin to appear.

5.4.3 Countermeasures

The DPA attack described in this chapter is only possible on a smartcard if the intermediate results are being used in a predictable way. Normally the program running on the main processor will not access these results, but the co-processor has to use them to perform the next step in the calculation. If the co-processor reads or writes its intermediate results in a predictable way, as we simulated above the implementation will be vulnerable to a DPA attack, a programmer has no influence on the implementation of the co-processor so there is not very much that can be done by the programmer apart from choosing a co-processor that is not vulnerable to DPA.

5.5 Conclusion

When implementing RSA on smartcards without having power analysis in mind, it will be very likely that the implementation is vulnerable to an SPA timing attack similar to the one described in section 5.3.

When countermeasures in the form of fake multiplications against this attack are implemented the possibility of a DPA attack as described in section 5.4 could be introduced if the co-processor uses the intermediate results in a predictable way.

As noted in section 5.4.3 the coprocessor for the smartcard should be chosen with care to prevent DPA attacks.

6 Conclusion

In this paper we've shown the results of our power analysis on smartcard algorithms using simulation. The PINPAS tool that took care of the simulation has proven itself very useful and, although it currently still is under active development, its potential is enormous. It can give information on the safety of smartcards early on in the design process, which can save money and valuable time.

The AES and RSA algorithms we have investigated using the PINPAS tool were both vulnerable to SPA and/or DPA attacks in their straightforward implementation. This shows it is very important to take these types of attacks into account when implementing algorithms on smartcards.

After we had added the *randomization* countermeasure to the AES implementation a DPA attack was no longer possible. Although not foolproof, the randomization counter measure does its job effective and efficient. And efficiency is often an important aspect to keep in mind when dealing with smartcard hardware.

Analysis on the RSA algorithm also gave some interesting results. The straightforward RSA implementation was vulnerable to an SPA attack. This attack was countered by the introduction of a fake multiplication. Depending on the type of coprocessor and the kind of hardware countermeasures included on it a DPA attack may still be possible. On a simple coprocessor which writes intermediate results of the RSA algorithm back to memory a DPA attack is probably possible on these intermediate values. A DPA attack on a coprocessor that uses an internal buffer for the intermediate results is generally more difficult if the coprocessor hardware doesn't leak too much information. In case of a coprocessor which stores its results in an internal buffer a fake multiplication instruction cannot be added to the algorithm because it would overwrite the internal buffer. This leaves the implementation vulnerable to SPA attacks, so the coprocessor should provide facilities to prevent this, for example by including a fake multiplication instruction.

The results have shown that power analysis attacks are definitely a risk factor that should be taken into consideration during the design of smartcards. During the design phase steps should be taken on software or hardware level to counter power analysis attacks. In this phase the PINPAS tool can be invaluable for analyzing proposed countermeasures and investigating the safety of the smartcard.

7 Acknowledgments

We are grateful to Erik de Vink for useful feedback and comments on an earlier version of this report.

References

[DaRij99] *The Rijndael Block Cipher*, J. Daemen, V. Rijmen
http://www.esat.kuleuven.ac.be/~rijmen/rijndael/rijndaeldocV2.zip

[GolTym03] *Multiplicative Masking and Power Analysis of AES*, J. Dj. Golic, C. Tymen
http://www.gemplus.com/smart/r_d/publications/pdf/GT03perm.pdf

[HVVVW03] *Pinpas: A Tool For Power Analysis Of Smartcards*, J. den Hartog, J. Verschuren, E. de Vink, J. de Vos, W. Wiersma. Proceedings of the Sec 2003.

[Koch96] *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, P. C. Kocher
http://www.cryptography.com/resources/whitepapers/TimingAttacks.pdf

[Koch98] *Differential Power Analysis,* P. Kocher, J. Jaffe, B. Jun, Advances in Cryptology - Crypto 99 Proceedings
http://www.cryptography.com/resources/whitepapers/DPA.pdf

[Mess99] *Investigations of Power Analysis Attacks on Smartcards*, T. S. Messerges, E. A. Dabbish, R.H. Sloan
http://www.eecs.uic.edu/~tmesserg/usenix99/paper_n.ps

[RijnAnim] *Rijndael animation*, E. Zabala
http://www.esat.kuleuven.ac.be/~rijmen/rijndael/Rijndael_Anim.zip

[RivShaAd78] *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,* R.L. Rivest, A. Shamir, L. Adleman
http://theory.lcs.mit.edu/~cis/pubs/rivest/rsapaper.ps

[Til99] *Fundamentals of Cryptology: A Professional Reference and Interactive Tutorial* H.C.A. van Tilborg, Kluwer Academic Publishers. ISBN 0-7923-8675-2

Appendix A

Initial AES C implementation

```
/**
  * rijndael.h, September 2003
  *
  * C implementation. As preparation of implemetation of
  * Rijndael in H8/300 assembler for use on smart cards.
  *
  * Authors: Gijs Hollestelle, Wouter Burgers
  */
#ifndef RIJNDAEL_H
#define RIJNDAEL_H

/**
  * Type defines; we only use uint8 and uint16 and no uint32,
  * since the H8/300 does not support that.
  */
typedef unsigned char  uint8;
typedef unsigned short uint16;

/**
  * Rcon table, used during generation of round keys.
  */
static uint8 Rcon[40] =
{
    0x01, 0x00, 0x00, 0x00,
    0x02, 0x00, 0x00, 0x00,
    0x04, 0x00, 0x00, 0x00,
    0x08, 0x00, 0x00, 0x00,
    0x10, 0x00, 0x00, 0x00,
    0x20, 0x00, 0x00, 0x00,
    0x40, 0x00, 0x00, 0x00,
    0x80, 0x00, 0x00, 0x00,
    0x1b, 0x00, 0x00, 0x00,
    0x36, 0x00, 0x00, 0x00
};

/**
  * The -forward- S-Box, used in the ByteSub transformation as
  * well as during the generation of round keys.
  */
static uint8 FSb[256] =
{
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5,
    0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0,
    0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC,
    0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A,
    0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0,
    0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B,
    0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85,
    0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5,
    0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17,
    0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88,
    0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C,
    0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9,
    0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6,
    0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E,
    0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
```

```
    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94,
    0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68,
    0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16
};

#endif

/**
  * rijndael.c, September 2003
  *
  * C implementation. As preparation of implemetation of
  * Rijndael in H8/300 assembler for use on smart cards.
  *
  * Authors: Gijs Hollestelle, Wouter Burgers
  */
#include "rijndael.h"

/**
  * Support function for generating the roundkey.
  * At the first call it generates the key for round 1, at the second
  * call the key for round 2, etc...
  *
  * @param roundkey   round key of previous round or the cipher key if this is
  *                           the first that time that _genkey is called.
  */
void _genkey( uint8 *roundkey )
{
  // Static variable to hold in what round we are.
  // Counting from 0, for easy index into Rcon.
  static int round = 0;

  uint8 tmp = roundkey[ 12 ];

  int i;
  for (i = 0; i < 4; i++)
  {
    if ( i == 3 )
    {
      roundkey[ i ] = Rcon[ 4*round + i ] ^ roundkey[ i ] ^ FSb[ tmp ];
    }
    else
    {
      roundkey[ i ] = Rcon[ 4*round + i ] ^ roundkey[ i ] ^ FSb[ roundkey[ i + 1 + 12 ] ];
    }

    roundkey[ i + 4 ] = roundkey[ i + 0 ] ^ roundkey[ i + 4 ];
    roundkey[ i + 8 ] = roundkey[ i + 4 ] ^ roundkey[ i + 8 ];
    roundkey[ i + 12] = roundkey[ i + 8 ] ^ roundkey[ i + 12];
  }

  // Increase round counter.
  round++;
}

/**
  * Support function for ShiftRow.
  *
  * @param state        pointer to an array of length 16.
  * @param row          which row to shift.
  */
void _rotate( uint8 *state, int row )
{
  int i;
  int j;
  uint8 tmp;

  for (i = 0; i < row; i++)
  {
     tmp = state[ row ];
     for (j = 0; j < 3 ; j++)
     {
       state[ row + j * 4 ] = state[ row + (j+1) * 4 ];
     }
     state[ row + 12 ] = tmp;
  }
```

```
}

/**
  * Support function for MixColumn.
  *
  * @param in         Input byte.
  */
uint8 _xtime( uint8 in )
{
  uint8 tmp;

  if ( in & 0x80 ) tmp = 0x1B;
  else             tmp = 0;

  in <<= 1;

  return in^tmp;
}

/**
  * ByteSub replaces every byte in the state variable with its
  * substitute from the forward S-Box.
  *
  * @param state        pointer to an array of length 16.
  */
void ByteSub( uint8 *state )
{
  int i;

  for (i = 0; i < 16; i++)
  {
    state[ i ] = FSb[ state[ i ] ];
  }
}

/**
  * The input state is ordered as a 4x4 matrix and shifted
  * as follows:
  *
  * from:                to:
  * 1  5  9 13            1  5  9 13
  * 2  6 10 14            2 10 14  2
  * 3  7 11 15           11 15  3  7
  * 4  8 12 16           16  4  8 12
  *
  * @param state        pointer to an array of length 16.
  */
void ShiftRow( uint8 *state )
{
  _rotate(state, 1);
  _rotate(state, 2);
  _rotate(state, 3);
}

/**
  * MixColumn performs a matrix multiplication. Every column from
  * the state, ordered in the above defined fashion, is multiplied
  * with the following matrix:
  *  2 3 1 1
  *  1 2 3 1
  *  1 1 2 3
  *  3 1 1 2
  *
  * This multiplication is performed over GF(2^8).
  *
  * @param state        pointer to an array of length 16.
  */
void MixColumn( uint8 *state )
{
  int i;

  for (i = 0; i < 4; i++)
  {
    uint8 tmp2 = state[ (i*4) + 0 ];
    uint8 tmp;
    uint8 tm;
```

```
    tmp = state[(i*4)] ^ state[(i*4) + 1] ^ state[(i*4) + 2] ^ state[(i*4) + 3];

    tm = state[(i*4) + 0] ^ state[(i*4) + 1]; tm = _xtime( tm ); state[(i*4) + 0] ^= tm ^ tmp;
    tm = state[(i*4) + 1] ^ state[(i*4) + 2]; tm = _xtime( tm ); state[(i*4) + 1] ^= tm ^ tmp;
    tm = state[(i*4) + 2] ^ state[(i*4) + 3]; tm = _xtime( tm ); state[(i*4) + 2] ^= tm ^ tmp;
    tm = state[(i*4) + 3] ^ tmp2; tm = _xtime( tm ); state[ (i*4) + 3 ] ^= tm ^ tmp;
  }
}

/**
  * AddRoundKey performs a XOR on the state array and the roundkey
  * array of the current round. The roundkeys are generated elsewhere.
  *
  * @param state        pointer to an array of length 16.
  * @param roundkey   pointer to an array of length 16,
  *                     representing the key for the current round.
  */
void AddRoundKey( uint8 *state, uint8 *roundkey )
{
  int i;

  for (i = 0; i < 16; i++)
  {
    state[ i ] = state[ i ] ^ roundkey[ i ];
  }
}

/**
  * Performs Rijndael encryption on an above defined input with a below
  * defined cipherkey.
  */
void Rijndael()
{
  uint8 input[16] =
  {
    0x32, 0x43, 0xf6, 0xa8,
    0x88, 0x5a, 0x30, 0x8d,
    0x31, 0x31, 0x98, 0xa2,
    0xe0, 0x37, 0x07, 0x34
  };

  uint8 key[16] =
  {
    0x2b, 0x7e, 0x15, 0x16,
    0x28, 0xae, 0xd2, 0xa6,
    0xab, 0xf7, 0x15, 0x88,
    0x09, 0xcf, 0x4f, 0x3c
  };

  // Perform initial AddRoundKey using the cipher key.
  AddRoundKey( &input, &key );

  // Perform 10 rounds.
  int i = 0;
  for (i = 0; i < 10; i++)
  {
    ByteSub( &input );

    ShiftRow( &input );

    // Skip MixColumn in round 10.
    if ( i != 9 )
    {
      MixColumn( &input );
    }

    _genkey( &key );
    AddRoundKey( &input, &key );
  }
}

int main()
{
  Rijndael();

  return 0;
}
```

Appendix B

Final AES C implementation

```
/**
  * rijndael.c, September 2003
  *
  * C implementation. As preparation of implemetation of
  * Rijndael in H8/300 assembler for use on smart cards.
  *
  * This version is modified to only work on global variables and
  * to use loop constructs using only GOTO and LABELS.
  *
  * Authors: Gijs Hollestelle, Wouter Burgers
  */
#include "rijndael.h"

/**
  * Loop variable for the 10 rounds of the algorithm.
  */
uint8 roundcount = 0;

/**
  * Various temporary and counter variables, used for various
  * purposes.
  */
uint8 i,j,tm,tmp,tmp2,tmp3;

/**
  * Text to be enciphered.
  * Length of 128 bits, 16 bytes.
  */
uint8 inputdata[16] =
{
  0x32, 0x43, 0xf6, 0xa8,
  0x88, 0x5a, 0x30, 0x8d,
  0x31, 0x31, 0x98, 0xa2,
  0xe0, 0x37, 0x07, 0x34
};

/**
  * Cipherkey.
  * Length of 128 bits, 16 bytes.
  */
uint8 key[16] =
{
  0x2b, 0x7e, 0x15, 0x16,
  0x28, 0xae, 0xd2, 0xa6,
  0xab, 0xf7, 0x15, 0x88,
  0x09, 0xcf, 0x4f, 0x3c
};

/**
  * Support function for generating the roundkey.
  * At the first call it generates the key for round 1, at the second
  * call the key for round 2, etc...
  */
void _genkey()
{
  // We need this later.
  tmp = key[ 12 ];

  key[ 0 ] = Rcon[ 4*roundcount ] ^ key[ 0 ] ^ FSb[ key[ 13 ] ];
  key[ 4 ] = key[ 0 ] ^ key[ 4 ];
  key[ 8 ] = key[ 4 ] ^ key[ 8 ];
  key[ 12 ]= key[ 8 ] ^ key[ 12 ];

  key[ 1 ] = Rcon[ 4*roundcount + 1 ] ^ key[ 1 ] ^ FSb[ key[ 14 ] ];
  key[ 5 ] = key[ 1 ] ^ key[ 5 ];
  key[ 9 ] = key[ 5 ] ^ key[ 9 ];
  key[ 13 ]= key[ 9 ] ^ key[ 13 ];

  key[ 2 ] = Rcon[ 4*roundcount + 2 ] ^ key[ 2 ] ^ FSb[ key[ 15 ] ];
```

```
  key[ 6 ] = key[ 2 ] ^ key[ 6 ];
  key[ 10 ]= key[ 6 ] ^ key[ 10 ];
  key[ 14 ]= key[ 10] ^ key[ 14 ];

  key[ 3 ] = Rcon[ 4*roundcount + 3 ] ^ key[ 3 ] ^ FSb[ tmp ];
  key[ 7 ] = key[ 3 ] ^ key[ 7 ];
  key[ 11 ]= key[ 7 ] ^ key[ 11 ];
  key[ 15 ]= key[ 11] ^ key[ 15 ];
}

/**
  * Support function for MixColumn.
  */
void _xtime()
{
  // Input en output zijn beide tm nu.
  if ( tm & 0x80 ) tmp = 0x1B;
  else             tmp = 0;

  tm <<= 1;
  tm ^= tmp;
}


/**
  * ByteSub replaces every byte in the state variable with its
  * substitute from the forward S-Box.
  */
void ByteSub()
{
  //int i;

I_LOOP_INIT:
  i = 0;

I_LOOP_BODY:
  inputdata[ i ] = FSb[ inputdata[ i ] ];

I_LOOP_EVAL:
  i++;
  if ( i < 16 )
  {
    goto I_LOOP_BODY;
  }

I_LOOP_EXIT:
  ;
}

/**
  * The input state is ordered as a 4x4 matrix and shifted
  * as follows:
  *
  * from:                to:
  * 1  5  9 13            1  5  9 13
  * 2  6 10 14            2 10 14  2
  * 3  7 11 15           11 15  3  7
  * 4  8 12 16           16  4  8 12
  */
void ShiftRow()
{
  // Rotate row 1, 1 left.
  tmp = inputdata[ 1 ];
  inputdata[ 1 ] = inputdata[ 5 ];
  inputdata[ 5 ] = inputdata[ 9 ];
  inputdata[ 9 ] = inputdata[ 13];
  inputdata[ 13] = tmp;

  // Rotate row 2, 2 left.
  // Needs 2 temporary variables.
  tmp = inputdata[ 2 ];
  tmp2= inputdata[ 6 ];
  inputdata[ 2 ] = inputdata[ 10];
  inputdata[ 6 ] = inputdata[ 14];
  inputdata[ 10] = tmp;
  inputdata[ 14] = tmp2;
```

```
  // Rotate row 3, 3 left.
  // So 1 right.
  tmp = inputdata[ 15 ];
  inputdata[ 15] = inputdata[ 11];
  inputdata[ 11] = inputdata[ 7 ];
  inputdata[ 7 ] = inputdata[ 3 ];
  inputdata[ 3 ] = tmp;
}

/**
  * MixColumn performs a matrix multiplication. Every column from
  * the state, ordered in the above defined fashion, is multiplied
  * with the following matrix:
  *  2 3 1 1
  *  1 2 3 1
  *  1 1 2 3
  *  3 1 1 2
  *
  * This multiplication is performed over GF(2^8).
  */
void MixColumn()
{
  //int i;
  //uint8 tm;

I_LOOP_INIT:
  i = 0;

I_LOOP_BODY:
  tmp2 = inputdata[(i*4)];

  tmp3 = inputdata[(i*4)] ^ inputdata[(i*4)+1] ^ inputdata[(i*4)+2] ^ inputdata[(i*4)+3];

  tm = inputdata[(i*4)]     ^ inputdata[(i*4)+1]; _xtime( ); inputdata[(i*4)] ^= tm ^ tmp3;
  tm = inputdata[(i*4)+1] ^ inputdata[(i*4)+2];   _xtime( ); inputdata[(i*4)+1] ^= tm ^ tmp3;
  tm = inputdata[(i*4)+2] ^ inputdata[(i*4)+3];   _xtime( ); inputdata[(i*4)+2] ^= tm ^ tmp3;
  tm = inputdata[(i*4)+3] ^ tmp2;                 _xtime( ); inputdata[(i*4)+3] ^= tm ^ tmp3;

I_LOOP_EVAL:
  i++;
  if ( i < 4 )
  {
    goto I_LOOP_BODY;
  }

I_LOOP_EXIT:
  ;
}

/**
  * AddRoundKey performs a XOR on the state array and the roundkey
  * array of the current round. The roundkeys are generated elsewhere.
  */
void AddRoundKey()
{
  //int i;

I_LOOP_INIT:
  i = 0;

I_LOOP_BODY:
  inputdata[ i ] = inputdata[ i ] ^ key[ i ];

I_LOOP_EVAL:
  i++;
  if ( i < 16 )
  {
    goto I_LOOP_BODY;
  }

I_LOOP_EXIT:
  ;
}

void Rijndael()
{
  // Perform an initial AddRoundKey using the CipherKey.
```

```
  AddRoundKey();

  // 10 rounds.
ROUND_LOOP_INIT:
  roundcount = 0;

ROUND_LOOP_BODY:
  ByteSub();
  ShiftRow();
  if ( roundcount != 9 )
  {
    MixColumn();
  }
  _genkey();

  AddRoundKey();

ROUND_LOOP_EVAL:
  roundcount++;
  if ( roundcount < 10 )
  {
    goto ROUND_LOOP_BODY;
  }

ROUND_LOOP_EXIT:
  ;
}

int main()
{
  Rijndael();

  return 0;
}
```

Appendix C

Initial AES assembler implementation

Since we're attacking in the initial AddRoundKey function of the algorithm, only that part of the code is shown. This is because it's the only part of the code that's relevant and including the entire code would take up a lot of space.
Full code can be obtained by compiling the C implementation above.

```
_AddRoundKey:
       push    r6;
       mov.w   r7,r6;
.L38:
       sub.b   r2l,r2l;
       mov.b   r2l,@_i;
.L39:
       mov.b   @_i,r2l;
       mov.b   #0,r2h;
       mov.w   #_inputdata,r3;
       add.w   r2,r3;
       mov.b   @(_key,r2),r2l;
       mov.b   @r3,r0l;
       xor     r0l,r2l;
       mov.b   r2l,@r3;
.L40:
       mov.b   @_i,r2l;
       add.b   #1,r2l;
       mov.b   r2l,@_i;
       cmp.b   #15,r2l;
       bls     .L39;
.L42:
       pop     r6;
       rts;
```

Appendix D

Mathematica RSA setup

Generate primes pB and qB

```
pB = 0;
qB = 0;
While[ ! PrimeQ[pB], pB = Random[Integer, {0,2^256}]]
While[ ! PrimeQ[qB], qB = Random[Integer, {0,2^256}]]

pB
  94610016142293966990213622776359968129777759268010521749610971678314340725047

qB
  47180809150759178366875245737440563269160443993346610683867174425597939851267
```

Calculate modulus nB

```
nB = pB * qB
  44637771153598167773066162246405467821773988380822035958657637645605083029547446
    10959784475216657372439740833334093876224861232285040341585382764121584549
```

Generate public key eB

```
eB = 0;
While[GCD[eB, (pB - 1)*(qB - 1)]  != 1, eB = Random[Integer, { 0, 2^512 }]]

eB
  15663237264967633104765501919935762246377477357783723520870966546133334298055491
    69215049428734337560982881480035597117095832052512635830708681715757266413

GCD[eB, (pB-1)*(qB-1)]
1
```

Calculate private key dB

```
dB = PowerMod[eB,-1,(pB - 1) * (qB - 1)]
  12714969383812823149661889466131423187768704105455288608002845556850552460936486 3
    37255678195249679891350220316744658492419314363394429227773382489440821 17
```

Appendix E

Initial RSA assembler implementation

```
Begin Body
// Copy input to _inputdata
.INPUTLOOP:
  mov.w @(_input,r2),r3;
  mov.w r3,@(_inputdata,r2);
  add.w #2,r2;
  cmp.w #64,r2;
  bne .INPUTLOOP;

  mov.w #0,r2;
// Repeat for all bytes of the key
.BYTELOOP:
  mov.w #128,r3l;
  mov.w #0,r4;
// Repeat for all bits in that byte
.BITLOOP:
  // Always square
  BIGSQR #_result,#_modulus;
  mov.b @(_secretkey,r2),r5l;
  and r3l,r5l;
  // Multiply only if bit is 1
  cmp.b #0,r5l;
  beq .SKIPMUL;
  BIGMUL #_result,#_inputdata,#_modulus;
.SKIPMUL:
// Loop control
  add.w #1,r4;
  shlr  r3l;
  cmp.w #7,r4;
  bls   .BITLOOP;
  add.w #1,r2;
  cmp.w #63,r2;
  bls .BYTELOOP;

  ext;
End Body
```