ELEC6032: Cryptography and Data Compression Coursework

Course:	Cryptography and Data Compression (ELEC6032)
Examiner:	Dr Srinandan Dasmahapatra
Student:	Dimitris Kouzis – Loukas (DKL105 - 21270031)
Date:	2 May 2006

ELEC6032: Cryptography and Data Compression Coursework

Course:	Cryptography and Data Compression (ELEC6032)
Examiner:	Dr Srinandan Dasmahapatra
Student:	Dimitris Kouzis – Loukas (DKL105 - 21270031)
Date:	2 May 2006

Table of contents

1. Elementary Number Theory in Cryptographic Applications

1.1. Last 6 digits of 5^{121212}

This number has $\log_{10} 5^{131313} = 131313 * \log_{10} 5 = 92$ digits. We will use modular exponentiation to calculate its 6 less significant digits $5^{131313} \bmod 10^6$.

We do some initial calculations:

$$5^{10} \bmod 10^6 = 9765625 \bmod 10^6 = 765625$$

$$5^{20} \bmod 10^6 = \left(5^{10} \cdot 5^{10}\right) \bmod 10^6 = \left(765625 \cdot 765625\right) \bmod 10^6 =$$
$$= 586181640625 \bmod 10^6 = 640625$$

$$5^{100} \bmod 10^6 = \left(5^{20}\right)^5 \bmod 10^6 = 640625^5 \bmod 10^6 =$$
$$= 107899495400488376617431640625 \bmod 10^6 = 640625$$

$$5^{500} \bmod 10^6 = \left(5^{100}\right)^5 \bmod 10^6 = \left(640625^5\right) \bmod 10^6 = 640625 \ \ldots$$

$$5^{62500} \bmod 10^6 = 5^{12500} \bmod 10^6 = 5^{2500} \bmod 10^6 = \left(5^{500}\right)^5 \bmod 10^6 =$$
$$= \left(640625^5\right) \bmod 10^6 = 640625$$

$$5^{125000} \bmod 10^6 = 5^{62500 \cdot 2} \bmod 10^6 = \left(640625 \cdot 640625\right) \bmod 10^6 =$$
$$= 410400390625 \bmod 10^6 = 390625$$

$$5^{100 \cdot 7} \bmod 10^6 = \left(\left(5^{100}\right)^2\right)^3 \cdot 5^{100} \bmod 10^6 = 390625^3 \cdot 5^{100} \bmod 10^6 =$$
$$= 390625 \cdot 640625 \bmod 10^6 = 140625$$

We have:

$$5^{131313} = 5^{125000+6313} = 5^{125000+2500 \cdot 2+1313} = 5^{125000+2500 \cdot 2+500 \cdot 2+313} = 5^{125000+2500 \cdot 2+500 \cdot 2+3 \cdot 100+10+3}$$

So, $5^{131313} \bmod 10^6 = 5^{125000} \cdot 5^{2500 \cdot 2} \cdot 5^{500 \cdot 2} \cdot 5^{3 \cdot 100} \cdot 5^{10} \cdot 5^3 \bmod 10^6$

But, $5^{2500} \bmod 10^6 = 5^{500} \bmod 10^6 = 5^{100} \bmod 10^6 \Rightarrow$

$$5^{131313} \bmod 10^6 = 5^{125000} \cdot 5^{100 \cdot 7} \cdot 5^{10} \cdot 5^3 \bmod 10^6 \Rightarrow$$

$$5^{131313} \bmod 10^6 = 390625 \cdot 140625 \cdot 765625 \cdot 125 \bmod 10^6 =$$
$$= 640625 \cdot 765625 \cdot 125 \bmod 10^6 = 515625 \cdot 125 \bmod 10^6 = 453125$$

(Based on: http://mathforum.org/library/drmath/view/66970.html)
(Checked with: http://www.math.umbc.edu/~campbell/NumbThy/Class/Programming/JavaScript.html)

1.2. Table lookup decryption

We don't want to use the primary factors of n = 851567 = 877 * 971. We fill a table with the letters and their respective cipher calculated with $m^{61} \bmod 851567$.

A	653462	H	639412	O	373327	V	796311
B	429305	I	572563	P	554438	W	223894
C	660923	J	326760	Q	434809	X	471549
D	266342	K	83193	R	779165	Y	722994
E	363516	L	774711	S	311976	Z	739934
F	7251	M	299564	T	447991		
G	303689	N	117318	U	511097		

Then we reverse-look-up and have the plaintext. For example:

722994 653462 117318 303689 447991 311976 363516 is…
Y A N G T S E

1.3. Two ciphers, prime encryption keys, common n

We have $c_A = m^{e_A} \bmod n$ and $c_B = m^{e_B} \bmod n$.
We calculate $c_A^{-1} = m^{-e_A} \bmod n$ and $c_B^{-1} = m^{-e_B} \bmod n$.

Lets find the $\gcd(e_A, e_B)$ using the classical Euclidean algorithm with a slight modification:

```
While $e_A = e_B$
    if $e_A > e_B$ then
        $e_A = e_A - e_B$
        $c_A = (c_A \cdot c_B^{-1}) \bmod n = m^{e_A - e_B} \bmod n$
        Recalculate $c_A^{-1} = m^{-(e_A - e_B)} \bmod n$
    else
        $e_B = e_B - e_A$
        $c_B = (c_B \cdot c_A^{-1}) \bmod n = m^{e_B - e_A} \bmod n$
        Recalculate $c_B^{-1} = m^{-(e_B - e_A)} \bmod n$
```

At the end we will have $e_A = e_B = 1$ because e_A and e_B are co-primes and of course, $c_A = c_B = m^1 \bmod n = m$!

e.g. if m = 12, n = 851567, e_A = 67 and e_B = 71 we have:

$c_A = 12^{67} \bmod 851567 = 131102$, $c_A^{-1} \bmod 851567 = 615100$
$c_B = 12^{71} \bmod 851567 = 329208$, $c_B^{-1} \bmod 851567 = 462445$

By applying the algorithm we get:

e_A	e_B	$c_A \bmod n$	$c_B \bmod n$	$c_A^{-1} \bmod n$	$c_B^{-1} \bmod n$
67	71	131102	329208	615100	462445
67	4	131102	20736	615100	151825
63	4	34092	20736	794641	151825
59	4	193674	20736	705893	151825
55	4	798107	20736	663652	151825
51	4	572144	20736	165152	151825
47	4	819398	20736	440965	151825
43	4	529887	20736	575361	151825
39	4	4584	20736	232026	151825
35	4	235561	20736	789153	151825
31	4	789526	20736	165136	151825
27	4	659329	20736	109189	151825
23	4	73008	20736	678018	151825
19	4	443528	20736	10078	151825
15	4	126508	20736	343493	151825
11	4	834982	20736	164460	151825
7	4	65994	20736	568292	151825
3	4	1728	20736	118766	151825
3	1	1728	12	118766	70964
2	1	144	12	573625	70964
1	1	12	12	70964	70964

The last 2 steps aren't necessary since we have m^1 in the 3rd step before the end.

Exceptions:

1. If m = 0 then $c_A = c_B = 0$.
2. $c_A^{-1} = m^{-e_A} \bmod n$ and $c_B^{-1} = m^{-e_B} \bmod n$ may not exist. In that case, c_A / c_B have common prime factors with n which means that g = gcd(c_A, n) $\neq 1$. Obviously if this happens we have accidentally factorized n! We set the prime factors p = g and q = n/p and we proceed in finding the decryption key for one of the two public keys.

The implementation of this algorithm can be found in the file ex1/main.c. The program verifies that it can successfully decode all the numbers from 0 to n-1 for predefined n, e_A and e_B.

2. Smart Cards and SunZe's Theorem

2.1. Non-faulty model

Chinese Remainder Theorem (CRT) (adapted to our case)

Given primes p and q, we calculate $s_p = q^{-1} \bmod p$, $s_q = p^{-1} \bmod q$ and $e_p = q \cdot s_p$, $e_q = p \cdot s_q$.

Given a system of congruent c_p, c_q of a number x modulo the primes p,q ($c_p \equiv x \bmod p$, $c_q \equiv x \bmod q$) we can calculate a solution x with the following relationship: $x = \left(c_p \cdot e_p + c_q \cdot e_q\right) \bmod pq$.

In our case, c_p and c_q are $c_p = m^e \bmod p$ and $c_q = m^e \bmod q$. By applying the CRT directly we can calculate $c = m^e \bmod pq$.

This implementation of RSA can be found in the file ex2/main.c.

We create a message m e.g. m=9234832. Then we initialize a signature structure (initSignature()) by calculating the public components of the signature (n and decryption exponent d) and then we create the signature (sign()) of the message using Chinese Remainder Theorem. The receiver then can verify (verify()) that the message matches the message extracted by the signature. If it matches then we have a genuine message otherwise it's fake.

2.2. Faulty model and breaking the cryptosystem

We have two signatures of the same message, $c \bmod pq$ and $c^* \bmod pq$. From the CRT, we know that: $c \bmod pq = c_p \cdot e_p + c_q \cdot e_q$ and $c^* \bmod pq = c_{perr} \cdot e_p + c_q \cdot e_q$

As a result, $diff = c \bmod pq - c^* \bmod pq = \left(c_p - c_{perr}\right) \cdot e_p$. But $e_p = q \cdot s_p$ and as a result, $diff = \left(c_p - c_{perr}\right) \cdot s_p \cdot q$. We can see that q appears as a factor in $diff$ but it's also one of the two prime factors of the public n. As a result $\gcd(diff, n) = q$. Then $p = n / q$ and then we can calculate e as usually and sign whatever message we want.

The function "sign()" described above takes a parameter that specifies if it's going to perform correct or faulty signature creation. If it's faulty, then a random bit of the 16 bit number c_p (because $2^{15} < p < 2^{16}$ and $c_p = x \bmod p$) gets inverted. Then CRC is being used to calculate the signature as usually.

Note that the fact that we invert a random bit of the 16-bit number c_p may result to a new value that is larger than p. For example for 3-bit numbers, if p = 5 and c_p = 2, inverting the MSB of c_p will result in c_{perr} = 6. In a highly optimized hardware implementation of CRT this fact may produce unpredictable behaviour. We assume that CRT is performed correctly for c_p in the full 16-bit range.

We store the non-faulty signature to the "original" variable and the faulty signature to the "signature" variable. Then we call the function "factor()" that calculates the GCD of n and the difference between the two signatures. This way, it extracts the two primary factors of n, p and q. If these factors are correct (match the predefined) it returns 1 else 0.

We check exhaustively this procedure with messages from 0 to 300000. This implementation is also included in the file ex2/main.c.

3. Small exponents and cracking RSA

3.1. Theoretical discussion

If we know $\phi(n)$ then we can calculate the decryption key d from the public key e:

$d = e^{-1} \bmod ((p-1)(q-1)) = e^{-1} \bmod (\phi(n))$, because n is the product of two primes p, q and as such, $\phi(n) = (p-1)(q-1)$.

3.2. Continued fraction algorithm implementation

The implementation of continued fraction algorithm for breaking RSA can be found in the file ex3/main.c

It creates a random test case and tries to break it using the algorithm described in the coursework. If the program gets executed with 1 or 2 as first parameter, then some predefined tests found in the literature[1,2] are being checked.

Random tests are being created by:
a) Using two predefined large primes p, q to create a 60-bits n
b) Randomly deciding the number of bits $\text{digits} \in [3, 20]$ for our d
c) Randomly creating a d, digits bits long
d) Testing that d is co-prime with $\phi(n) = (p-1)(q-1)$
e) Calculating the encryption key e as usually for the RSA
f) If any of the tests fails, then we go to step c

3.3. Applicability probing

A batch file (batch.sh) was created that runs the program 1000 times and creates a log file (log.txt) with the results for each execution. This file was analyzed using MS Excel and the results are presented in the following page.

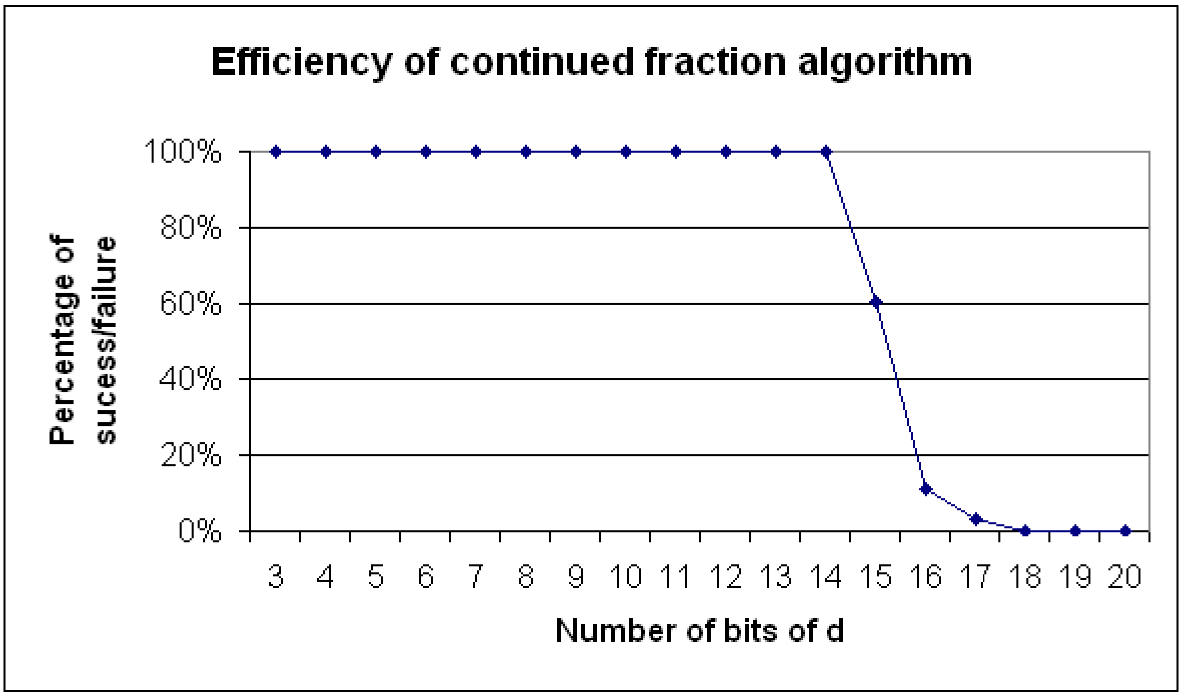
We can see that the theoretical upper limit of $\log_2(n) / 4 = 15$ bits is not 100% accurate. In fact if d is $(\log_2(n)/4)-1=14$ bits long we can almost certainly find d. For larger key lengths we can find d with a decreasing probability. For keys longer than 16 bits the probability is practically zero.

It must also be noted that our tests for binary length of d: 3 and 4 are not enough. We use a single value for n to generate all the tests and as a result, the number of 3-bit (only 5) and 4-bit (only 11) d's that are valid is very limited.

It's very strange that this version of the algorithm and more specifically the test about "integer roots r1, r2", does fail in the simple test case described in [Wiener89] (run: ./ex3 2).

[1] "Continued Fractions and RSA with small secret exponent", A. Dujella, 2004
[2] "Cryptanalysis of Short RSA Secret Exponents", J. Wiener, 1989

# bits of d	Occurrencies	Successes		Failures		Errors		Sum	
3	44	44	100%	0	0%	0	0%	44	100%
4	59	59	100%	0	0%	0	0%	59	100%
5	62	62	100%	0	0%	0	0%	62	100%
6	58	58	100%	0	0%	0	0%	58	100%
7	59	59	100%	0	0%	0	0%	59	100%
8	54	54	100%	0	0%	0	0%	54	100%
9	43	43	100%	0	0%	0	0%	43	100%
10	49	49	100%	0	0%	0	0%	49	100%
11	52	52	100%	0	0%	0	0%	52	100%
12	60	60	100%	0	0%	0	0%	60	100%
13	64	64	100%	0	0%	0	0%	64	100%
14	60	60	100%	0	0%	0	0%	60	100%
15	53	32	60%	21	40%	0	0%	53	100%
16	55	6	11%	49	89%	0	0%	55	100%
17	62	2	3%	60	97%	0	0%	62	100%
18	55	0	0%	55	100%	0	0%	55	100%
19	50	0	0%	50	100%	0	0%	50	100%
20	62	0	0%	62	100%	0	0%	62	100%



* Note: n's length is 60 bits and as a result length(n)/4 = 15

4. Timing Attacks on Cryptosystems: RSA

Here we implement Kocher’s timing attack. In order to be in accordance with the definition of the coursework we have the same numbering for chapters. Most of the content, though, is presented in the forth chapter.

4.1. Square and Multiply

The implementation of the square and multiply algorithm that we use is straightforward and occurs directly from the definition. Note that the calculation is being performed from the LSB of the exponent to the MSB in contrast to the algorithm presented in Kocher’s attack.

```
function r = Exponentiate(b, a, n); // Calculates b^a mod n
r = 1;
x = a
// Note: Assuming b[0] -> MSB
For every bit i of the exponent b {
  If (b[i]) {
    r = (r * x) mod n
  }
  x = (x * x) mod n
}
```

This implementation has the interesting property that for given n and a and fixed length of b, the only think that changes the final delay is the modulus multiplication inside the “if” block.

This is good and bad at the same time. It’s bad because it reduces the variance of the time because b-derived decisions propagate slower than in Kocher’s version. It’s good because our measures are more deterministic and variances comply better to the formula $Var(e)+(w-b)Var(t)$ for correct and $Var(e)+(w-b+2)Var(t)$ for wrong guess. This is because there are less correlations between t_is and as a result “…the modular multiplication times are effectively independent from each other…”. In the limit of large number of samples, there is no difference but our samples and processing power is limited so this gives better results with fewer samples than Kocher’s. There is also another very important advantage presented in section 4.4.2.1.

4.2. Simulated time measure

The timing model of the square and multiply algorithm is simple: we assume that only the modular multiplication consumes time. There have been developed many different timing models for the modular multiplication. The simplest that has been used for most of the development is this:

Time = Population count of the smallest operand of the multiplication.

The main reason that this has been used is that it’s quick. It’s also in good accordance with a theoretical ASIC multiplier as described in section 4.4.1.3.

It must be noted that all the models have been used during development and that the attack was developed using the insights gained by analyzing data from these models.

This is important because if we had gone step by step in this exercise and we only had one source of timing information for the development step 3 it is highly likely that in step 4 we would have to refine the algorithm quite a lot.

4.3. Kocher's attack

In order to implement Kocher's attack we need to model the whole crypto-system. We create the keys with the create_keys.c. We use two 128 bit primes and a random odd 249-bit decryption exponent. n is 252 bits long. The numbers that I use are the following:

```
p = 25525635435900842730349748303929424117
q = 259965242284515788826732110240207250949
n = 663577800056018436974822727090414862381501409992065761671325917
         1657271737033
phi = 6635778000560184369748227270904148623529523222200240985156177
         313113135061968
e = 638411998542464493022786090070785953192786232568308668940017047
         0081602434929
d = 705026298859753003824815613974422379277105777162354528745769699
         766147221969
```

By using these numbers I create random non-zero 252-bit plaintexts and measures for their times. This is being done with the `create_measures.c` file and the `create_measures` executable. `create_measures` is being configured for the appropriate modular multiplication algorithm by the `compile_create_measures` script. For example typing:

```
compile_create_measures simple
```

will create a `create_measures` executable for the simple timing model (mul_mod_simple.c). The output of this file is being stored on a file that is suggested to have the following name: mes_XXXX_err_RR.txt where XXXX is the name of the timing model used e.g. simple and RR is the amount of error that the file has.

There were two needs that made the need of intermediate files important:

- The one is the need for repeatability during debugging. We could of course give the random seed number as argument to the program but it would be awkward.

- The most important problem is that the generation of these data requires a lot of time (up to 20 minutes / 5000 encodings in a small machine for a complex Modular Multiplication Model). As commented in the paper, at least 4 times the same processing power would be required from the attacker but better 4 than 5. Additionally during debugging we want quick tests. The use of these intermediate files accelerated amazingly the design process.

These data files are being examined by the `main` program (`main.c`) that is the implementation of Kocher's algorithm. This is also reconfigurable with the same way as the `create_measures`. For example by typing:

```
compile_main simple
```

we create an attack executable (`main`) for the simple timing model.

To summarize the whole process we give an example:

```
linux:~/crypto/ex4 # ./compile_create_measures simple
    gcc -g -lgmp -lm -O2 -o create_measures create_measures.c
    square_and_multiply.c mulmod_simple.c
linux:~/crypto/ex4 # ./run_measures simple
    ./create_measures >  mes_simple_err_00.txt
linux:~/crypto/ex4 # ./compile_main simple
    gcc -g -lgmp -lm -O2 -o main main.c square_and_multiply.c
    mulmod_simple.c
linux:~/crypto/ex4 # ./run simple
    ./main mes_simple_err_00.txt
```

As you can see there are two additional script `run_measures` and `run` that accelerate common tasks. The `run` script takes two arguments, the timing model and the error level. If error level is omitted 0 is assumed.

With this attack we don't consider the length of the key because in most of cases it's known. Additionally we could find it because the variance will start increasing after the continuous decrease of the last bits of the key. Only this bits' value is what we are searching.

4.4. Putting it all together

Here we present the timing models (section 4.4.1), the attack and its advanced features (section 4.4.2), observations on the results of the analysis (section 4.4.3), protection methods (section 4.4.4) and possible improvements (section 4.4.5).

4.4.1 Modular multiplication's models

These files where tested by comparing their results with GMP's routines. All the functions return a `run_info_t` structure that contains information about the run (time, iterations, multiplications etc.) The test can be created by compiling `mulmod_test.c` with the appropriate `.c` file for the model of the multiplication.

e.g. `gcc -g -lgmp -lm -O2 -o test mulmod_test.c mulmod_simple.c`

4.4.1.2 Simple model of modular multiplier (mulmod_simple.c)

This simple model returns the population count of the smallest operand of the multiplication. Its main feature is that it's fast. It describes a highly sequential implementation of modular multiplication that uses a (slow) addition each time it finds a '1' in one of the two operands. At the beginning of its operation it swaps the two operands in order to use the smallest for controlling the sequence and the largest for the additions, thus accelerating the calculation.

4.4.1.2 Discrete modular multiplier (mulmod_discrete.c)

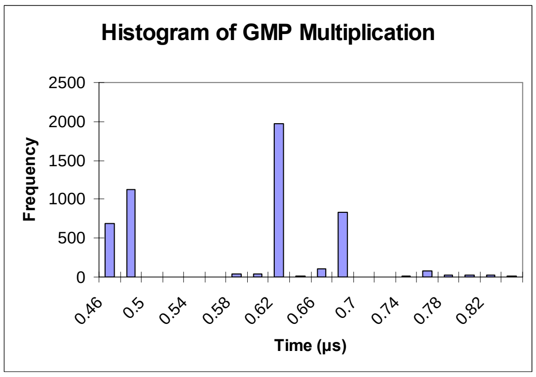
This implementation uses a (assumed) very low cost hardware multiplication (n x n = 2n bits). The data dependant overhead occurs from the analytic calculation of the modulus that is actually a bare division[3]. The only reason to use this instead of the next one is that you may share these general purpose components e.g. on a general purpose computer.

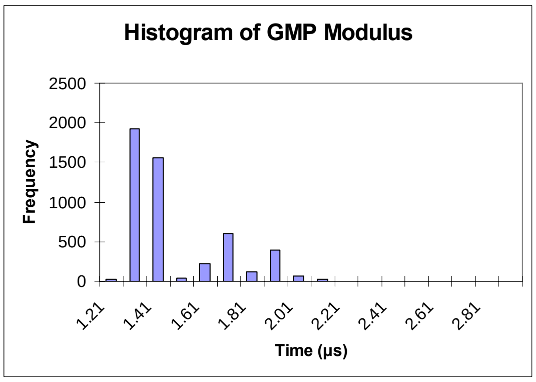
4.4.1.3 Galois Field (2^n) ASIC modular multiplier (mulmod_galois.c)

This multiplier was based on a Xilinx's Application Note[4]. It models a fast multiplier that performs multiplication and modulus at the same time. It performs radix-2 multiplication step-by-step by adding one multiplicand if the other's bit is 1. If this addition is larger than n, it subtracts n from the result. The benefit of this implementation is that it can be implemented with n+1 units instead of n^2 of most alternative implementations and of course it's fast.

4.4.1.4 Real Modular Multiplication (mulmod_real.c)

I didn't try to make a model GMP's implementations on my PC. I did some profiling on them and it shown as apparent from the following histograms, that they are very complex and as described in the documentation they use different algorithms for different ranges of numbers.





Instead of modeling them I used the most accurate model that could ever exist; the same environment. Eve is supposed to have exactly the same software/hardware as Bob and is making measurements. The advantage of this approach is that it's actually cheaper than the developing a model for the machine and that it's universal. The same attack (on source or executable) on different machines similar to Bob's will produce good results. It might be also possible to have execute permission on the server you attack. Under this circumstance this "model" also simulates the Bob's workload.

Making more accurate time measurements by summing the time of a large number of modular multiplications and then dividing by their count DIDN'T make the attack more successful as expected. For example by measuring 10 modular multiplications (following 2 dummy ones to optimize predictions) we ended up with a very bad performance of the attack: "248 bits, 145 errors (58.5%)" which means that the attack was a little bit biased on predicting wrong values! Note that gcc optimization level should be better turned off in order to count time.

[3] "Division Algorithms and Implementations", S. Oberman, M. Flynn, IEEE transactions on computers vol. 46, No. 8 , August 1997
[4] "XAPP371: CoolRunner-II CPLD Galois Field GF(2m) Multiplier", September 26, 2003, Xilinx

4.4.1.5 Constant time

If an implementation provides constant time execution, the attack can not be applied. As a result this case is not being considered.

4.4.2 The attack and its advanced features

Our implementation of the attack is quite sophisticated using many improvements inspired by Kocher's paper. Even thought, it gives average results. Perhaps Kocher has things that he doesn't say in his paper that make his basic implementation give so good results as he claims.

4.4.2.1 Step by Step Square and Multiply

We implemented a partial implementation of square and multiply which is able to calculate the exponentiation and the time up to the n-th step and then given more bits for the exponent it can very quickly calculate the new exponent and accurately its new time.

For example given the exponent `"111010001"` we call

```
sam_step a = initSamStep(c, n, error_level);
square_and_multiply_step(a, "111010001", 9);
```

The first call initializes a SquareAndMultiply step structure (`sam_step`) with the given c and n and the desired error level. The second call runs the square and multiply algorithm using the 9 bit of the exponent passed as parameter.

The `sam_step` structure has as members all the information we may needed. Apart from the result (`a->r`) it includes a `run_info_t` structure (`a->d`) that has all the timing information about the square and multiply algorithm up to this execution step.

Then by simply calling:

```
square_and_multiply_step(a, "10", 2);
```

we update the `sam_step` structure a as if the exponent was `"10111010001"` i.e. we added two more Most Significant Bits. The performance overhead of this function is the cost of two iterations instead of complete key's.

By using the `copySamStep` function we can reduce the overhead even more. E.g. to calculate all the variants of the key a before, we would have just to write:

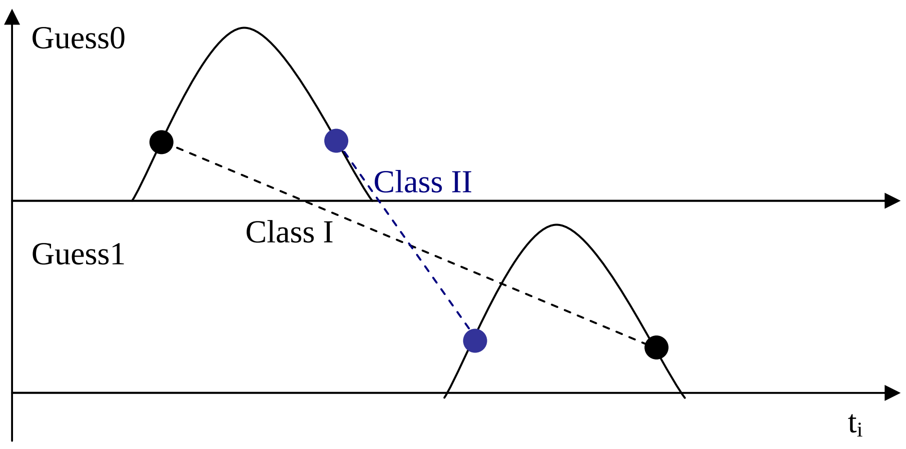
```
A00 = copySamStep(a);
square_and_multiply_step(A00, "00", 2);
A01 = copySamStep(a);
square_and_multiply_step(A01, "01", 2);
A10 = copySamStep(a);
square_and_multiply_step(A10, "10", 2);
A11 = copySamStep(a);
square_and_multiply_step(A11, "11", 2);
```

These 11-bit calculations get completed with 9 + 4 * 2 = 17 iterations instead of 4 * 11 = 44. The savings are very significant with very large keys (e.g for the same case

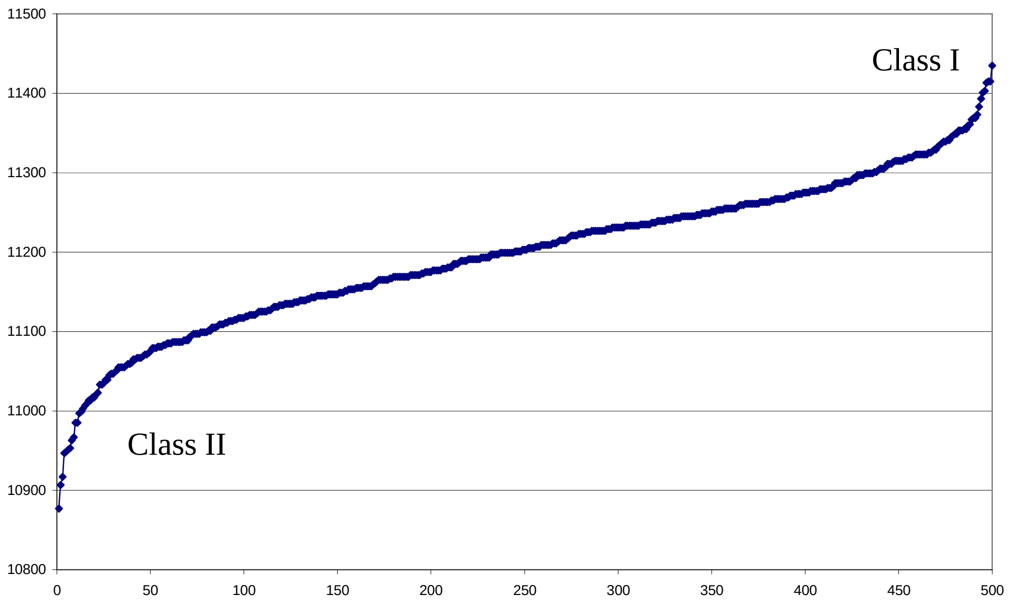
with 255 bit key there would be: 253 + 4 * 2 = 261 instead of 4 * 255 = 1020 iterations = 390% performance increase).
This optimization wouldn't be realizable (without affecting accuracy) if we implemented square and multiply algorithm that calculates from exponent's MSB to LSB. Our implementation of the attack works anyway. This just increases performance.

4.4.2.2 Elite filter

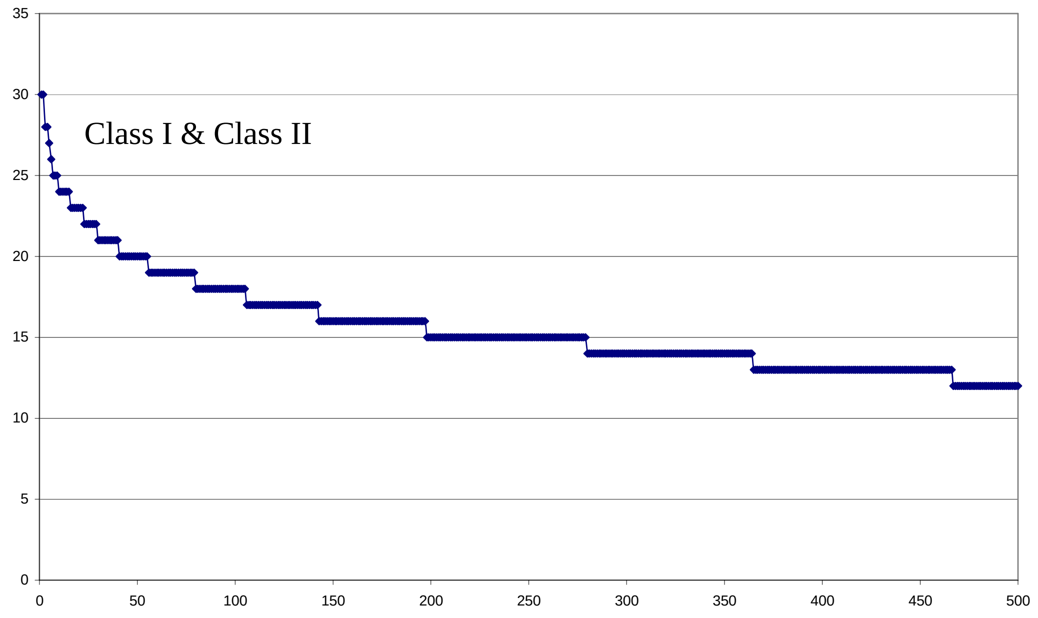
Given a almost normal distribution of calculation time when the guess for the next bit is 1 and a normal distribution when the guess is 0, we know that definitely that on average Guess1's value will be larger than Guess0's. This is because one additional modular multiplication is needed when we have a guess of 1.



Despite that, there are some ciphers that have at the same time very fast Guess0 and very slow Guess1 (Class I) or very slow Guess0 and very fast Guess1 as illustrated in the figure above. At the figure bellow we can see the difference *diff* = Guess1-Guess0 for each of the 500 sample ciphers. It is obvious that those two classes actually exist.



If we calculate $\left|diff - \overline{diff}\right|$ and we sort the results we have the two extreme classes collapsed.



By using an elite subset of samples we can have a major improvement on the difference of variances between 1 and 0 guesses for the bit under consideration. This difference isn't biasing but just a filtering of the noisy-don't care ciphers that certain bit.
It must be noted that using 50 top samples of the $\left|diff - \overline{diff}\right|$ is better than just selecting 25 from the top and 25 from the bottom of $diff$ because this criterion performs better even with asymmetric distributions of Guess0 and Guess1.

Examples: We can see in the following tables values for different cases square and multiply timing models. g0 means the variance of guess0, g1 means the variance of guess1 and fx means the variance of the fixed part. In all tables two cases are being considered: The fixed "010001" where the correct guess is 1 and the fixed "10001" where the correct guess is 0. The highlighted parts are the digits that differ between 0 and 1 guess.

Simple		
	Before	Filtered
g0: 0010001 g1: 1010001		
g0	176.523214	177.392164
g1	176.**255937**	17**6.894773**
fx	176.881205	177.194924
g0: 010001 g1: 110001		
g0	17**6.881205**	17**4.924763**
g1	177.021947	175.354602
fx	177.303097	175.490683

Discrete		
	Before	Filtered
g0: 0010001 g1: 1010001		
g0	859079.482203	446.375104
g1	859079.48**1873**	44**4.560704**
fx	859079.482685	449.130967
g0: 010001 g1: 110001		
g0	859079.482**685**	43**8.452123**
g1	859079.482867	439.983208
fx	859079.482868	439.890916

Galois		
	Before	Filtered
g0: 0010001 g1: 1010001		
g0	1623.484583	1655.823146
g1	162**2.683886**	165**4.675993**
fx	1624.663431	1657.913911
g0: 010001 g1: 110001		
g0	1624.663431	1549.093634
g1	162**3.251916**[*]	15**36.937299**[*]
fx	1627.952895	1551.809016
		[*]This guess is wrong.

As we can see the elite filter has very strong effect on the variance and increases the difference in all the cases.

4.4.2.3 Lookahead – variable window

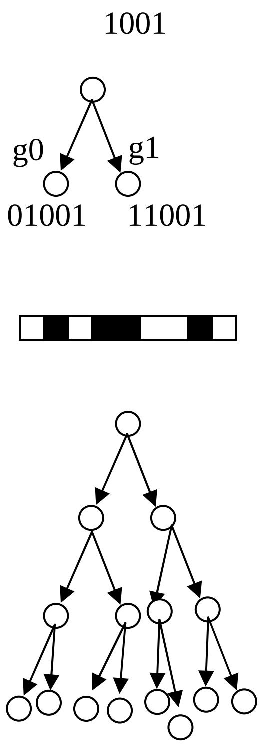
When you have a fixed guess e.g. 010001 and you are trying to guess the next bit, you can do it by considering the next MSB thus having two guesses; **0**010001 and **1**010001. You can also consider the two MSB's and thus having two groups of guesses… {**00**010001, **10**010001} and {**01**010001, **11**010001}. If you see that the first group has somehow better members e.g less average variance or least minimum variance than the second, you can conclude that the bit under consideration is 0 otherwise it's 1.

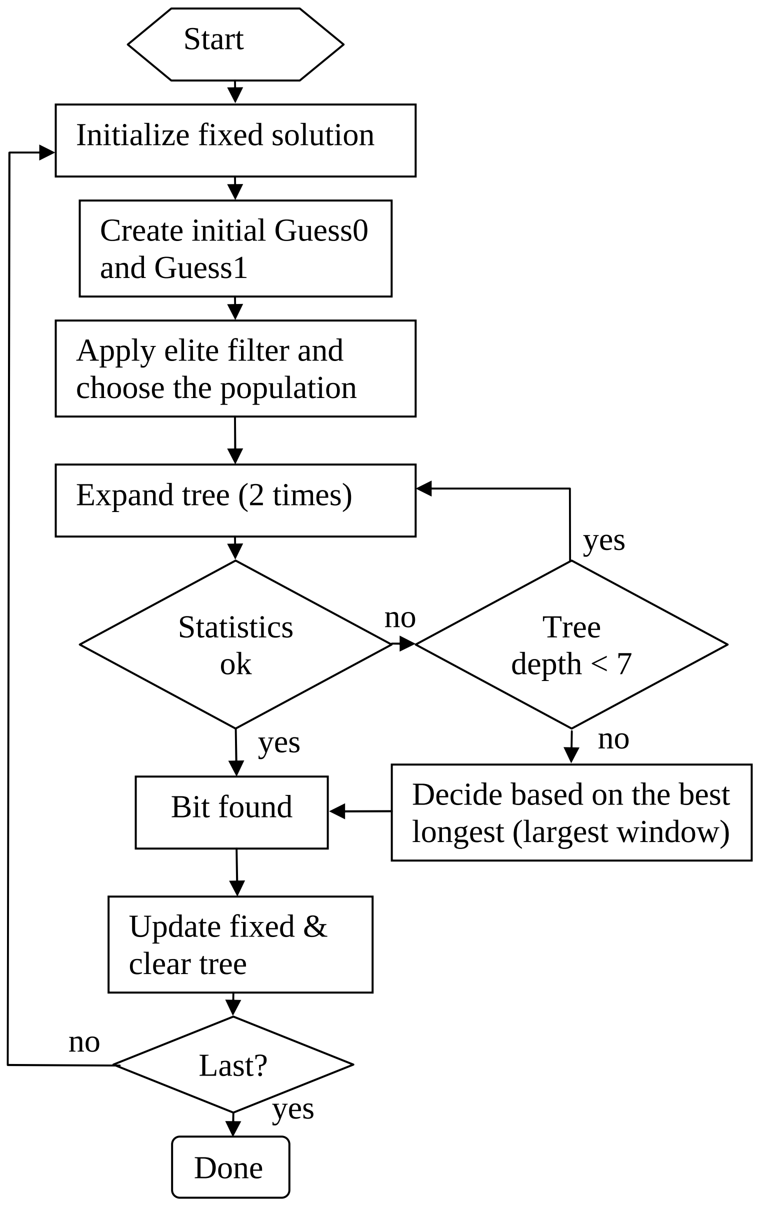
This way we can lookahead and gain more knowledge about it. This is actually an error preventing mechanism (instead of error correcting) because as stated in the paper, if a wrong guess has been made, the measures will soon become uncorrelated.

Even more, if one is not confident about his guess about a specific bit he can increase the window (the number of MSBs he considers) until he has a statistically acceptable amount of certainty.

4.4.2.4 The complete algorithm

The flowchart for the complete algorithm can be seen in the following figure.




Start
Initialize fixed solution
Create initial Guess0 and Guess1
Apply elite filter and choose the population
Expand tree (2 times)
Statistics ok
no
Tree depth < 7
yes
yes
no
Bit found
Decide based on the best longest (largest window)
Update fixed & clear tree
no
Last?
yes
Done

Initially the fixed (known) part of the solution gets initialized. Then we crate the two candidate solutions g0 and g1. We calculate the times for each of them for each cipher. Based on these measurements we form the elite population that maximizes the variance.

Then we expand the tree so that it has a depth of 3. (This and the following is just a policy that can be easily adapted to other specifications). We calculate the variance of depth 1 for 1 and 0 population. Let them be t00 and t01 respectively. Then we calculate the variances for the other two depths as well. Let them be t10, t11 and t20, t21. If t00>t01 two votes go to 1 (least variance) else two votes go for 0. Then for the variances of depths 2 and 3 the same procedure applies but just with one vote instead of two. By voting, depth 1 has the greatest weight and depth 2 and 3 can at most result in equal voting result between 0 and 1 and never beat depth 1.

If after this the voting is still 0, then the tree gets expanded by two. Then t00 and t01 remain the same but t10, t11 and t20, t21 represent the 4th and the 5th depth respectively. This continues till there is a decision or the tree depth gets more than 7 levels deep. In the latter case we decide according to the best of the deepest level.

When a bit is found, the tree gets cleared, fixed solution gets updated and the procedure restarts.

4.4.3 Observations on results

In the following sections we summarize our observations on the performance of the attack as a function of different parameters.

4.4.3.1 Dependence on timing model

The dependence on the timing model has been studied by using an elite population of 800 ciphers and 0% error level data. In the following table we can see the results.

Timing model	Bits run	# Errors	Error probability	# Expands
simple	248	7	2.8%	2
discrete	78	7	9.0%	5
Galois	38	8	21.1%	12
real	248	118	47.6%	29

It must be noted that discrete and Galois models are very slow and as a result a partial run got executed. This also means that their results are a little bit pessimistic because the first bits are harder to guess accurately than the latter (more specifically after 50% (in our case 130) of bits found).

We can also see that the real model has a performance as low as 47%. This means almost random results! This happens because of the complex timing model of GMP's multiplication and modulus functions. There are two key causes of this inerrability:

1. Inaccurate measurements. Efficiency of the modular multiplication varies with OS's load. For example the first about 100 iterations from each dataset of a real measurement is useless because it's at least 12% slower than all the other data.
2. Inaccurate model. Because the algorithm we run is different when we generated data and when we run the attack, we have an inaccurate model of the real machine. Unfortunately the calling environment defines many parameters that affect the performance of modular multiplication.

Summarizing our findings, we can conclude that systems that have a great variance on timing are more vulnerable to timing attacks. Of course this isn't something new. The important thing is that general purpose controllers that are being used explicitly for cryptography are the most vulnerable. This is because they show a very predictable behavior, they don't have optimized operators that limit the variance by increasing performance of the calculations and they usually have very light operating systems to obfuscate the timing data. The general purpose servers that use RSA implementations are the most secure because the fact that they run several tasks at the same time makes the final timing measurements almost useless. In average vulnerability region are ASICs and other special cryptographic controllers that don't take any special actions to prevent from timing attacks.

4.4.3.2 Dependence on elite population count

In the following table we can see the dependence of error probability and elite population for simple model with 0% noise:

Elite population	Error probability
500	5.6%
800	2.8%
1000	3.2%

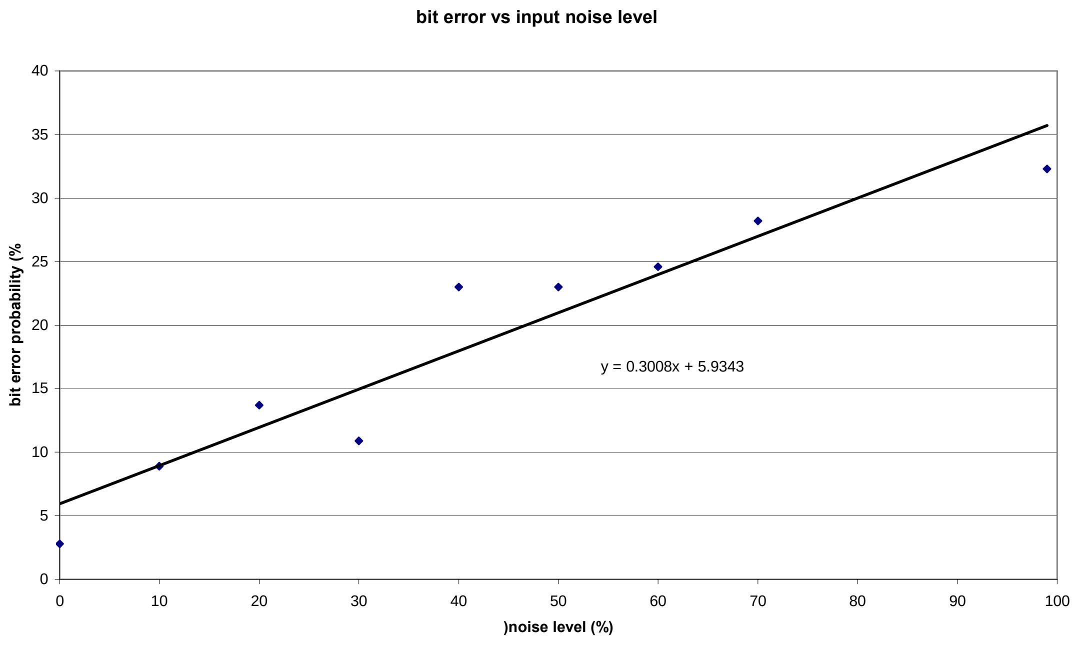
Of course the study of the dependence on population count is far from complete but we see what we expected. There is a tradeoff on the population count. If it's too little then we have a little population of over-adapted ciphers that are not sufficient for giving accurate measurements for guess0 and guess1 variances. If they are too many then they allow a lot of noisy-don't care solutions that reduce the difference of variances between guess0 and guess1.
Elite population count could be a nice parameter for an adaptive approach. For example one could run the first e.g. 20 bits and if the estimated error levels aren't sufficient, adjust the elite population count till satisfied. Unfortunately modifying it and re-calculating adds a lot of performance overhead.

4.4.3.3 Dependence on noise levels

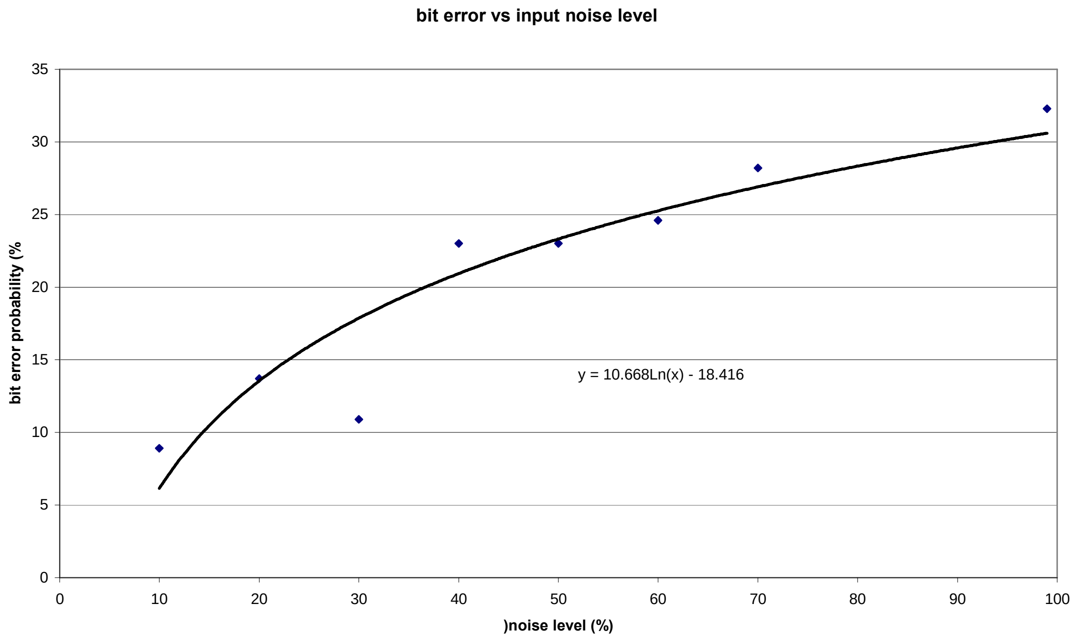
The dependence on noise has been studied with the simple timing model only using 800 as elite population count. In the following table we can see the results. We may note the importance of tree expanding that increases with noise.

Input noise level	# Errors	Error probability	# Expands
0%	7	2.80%	2
10%	22	8.90%	9
20%	34	13.70%	9
30%	27	10.90%	9
40%	57	23.00%	17
50%	57	23.00%	8
60%	61	24.60%	19
70%	70	28.20%	12
99%	80	32.30%	17

In the following figure we can see the average error probability on a bit as a function of the noise of the input samples. As we can see we have an increase of 3% on the error probability for each 10% of added noise.



Of course this function is not really linear because it must asymptotically approach 50% (completely random). A logarithmic dependence as we can see in the next figure fits better the data.



4.4.4 Protection methods

The most powerful method for protecting from this attack is to modify the protocol a little and use different decryption exponents. Then the attack definitely will be unable to work.
Another method is to try to make the Square and Multiply operation constant time. This is on the one hand relatively difficult to achieve and on the other it will negatively affect system's performance.

Another way is to add some delays with data depended patterns which will obfuscate the attacker. These delays can be rare enough in order not affect the performance significantly and will also prevent the system from power and tempest attacks.

4.4.5 Improvements

The best way to improve this attack is to add an error correcting feature. It is difficult to know at a given time if the bit you are trying to find is correct or not correct. The data that are being used may be misleading. There are many ideas from different areas like adaptive systems, artificial intelligence, pattern recognition and data mining that can be applied to solve this problem.

Many of them include doing many experiments with different partitions of the ciphers more or less cleverly selected and combining their results. The lookahead size and the elite population count that can be also adaptively changed to give more accurate measurement for a given bit.

The good fact about this problem is that if there are errors you will see the variance increasing after a while. This means that in a margin of e.g. 10 bits (depending on the modular multiplication algorithm and the data) you can know that you have done an error. At that moment you can e.g. run simulated annealing on the last e.g. 20 bits to see if you can get a better solution.

Another good solution proposed on the Kocher’s paper is to have a pool of many possible keys developed in parallel and sort them according to their fitness.

Generally one can (as always in signal processing) increase the dimensions of the problem and get better results. If the increase on the development and computational effort is worth the improvement is the biggest problem.