# Attack by Faults: DFA attacks on RSA

Mohammed ALMANSOORI, Eiman ALSHEBLI
and Omar BANI HASHIM

Emails: mohammed.almansoori@ensimag.imag.fr, alsheble@ensimag.imag.fr
and banihaso@ensimag.imag.fr

## 1 Abstract

Since the attacks by faults considered as a powerful way to attack the RSA. The attacks describe in this paper are based on corrupting the public modulus $N$ [BCG08][BCDG09]. We describe here the implementation of two attacks on the modular exponentiation part of the RSA system.
The implementation details and results were obtained using our own implementaion of the attacks using the (GIVARO-3.2.13 library).Since the two attacks cover both the iterative and the recursive modular exponentiation algorithms, which raises the concern for the need for protecting the public key

## 2 Introduction

Injection faults is considered as a powerful way to attack systems in order to recover private inforamation. There are many attacks implemented using this method. In this paper, we will describe the theory and implementation sides of the two Differential Faults Analysis (DFA) Attacks.
The first attack is described in the paper (Perturbating RSA Public Keys: An improved Attack)[BCG08]. While, The second one is described in the paper (Faults Attacks on RSA Public Keys: Left-To-Right Implementation are also Vulunerable) [BCDG09]. Our Implementation is based on those two papers.
This paper consists of four sections. The theory section includes a general background of the RSA system and the information for the attack requirments. The second section, which is the implementaion section is about the algorithm impelementation of the two attacks. Before the end we present the results of both attacks. And finally, the conclusion of this paper.

# 3 Background

Before starting to talk about the modular exponentiation, we need to consider the following notations. We will assume that $N$ is the public modulus and it is the product of two large prime numbers $p$ and $q$. Also we will assume that the length of $N$ is $n$. Let $e$ be the public exponent and $d$ as the private exponent. Both $p$ and $q$ are coprime to $\varphi(N) \equiv (p-1)\cdot(q-1)$, where $\varphi(.)$ denotes Euler's totient function. The exponents $e$ and $d$ are linked together by the equation $e \cdot d \equiv 1 mod \varphi(N)$s. Two operations can be performed using the private exponent $d$. The first operation is the RSA Decryption in which you decrypt a ciphertext $C$ by computing $\widetilde{m} \equiv C^d mod\, N \equiv C^{\sum_{i=0}^{i=n-1} 2^i \cdot d_i}$. $mod\, N$ where $d_i$ stands for the i-th bit of $d$. The$\widetilde{m}$ will be equal to $m$ if there were no errors during the computation, transmission or decryption of $C$. The second operation is the RSA signature. To sign a message $m$ with a sign $S$, the signature should be as $S = \dot{m}\, mod\, N$ where $\dot{m} = \mu(m)$ for some hash and /or deterministic padding function $\mu$. To know that the signature $S$ is validated, we need to check that $S^e \equiv \dot{m}\, mod\, N$.

# 4 Theory

## 4.1 Modular Exponentiation

In this section, we will talk about the implementation of the two different binary exponentiation algorithms that are used often for computing RSA modular exponentiation $\dot{m}^d mod\, N$. The exponent $d$ in the previous equation is expressed in binary form as $d = \sum_{i=0}^{n-1} 2^i \cdot d_i$.

### 4.1.1 Iterative Algorithm (Right to Left)

This algorithm calculates the modular exponentiation by scanning iteratively the bits of the private exponent $d$ from the least (LSB) to most significant bit (MSB). This was the reason for calling it the Right to left algorithm. This algorithm is considered as the most used algorithm to compute the modular exponentiation.

Iterative Algorithm "Right to Left modular exponentiation"

INPUT: $m$, $N$, $d$
OUTPUT: $A \equiv m^d mod\, N$

```
1 : A := 1;
2 : B := m;
3 :  for i from 0 upto (n − 1)
4 :    if (d_i == 1)
5 :      A := (A · B) mod N ;
6 :    endif
7 :    B := B^2 mod N ;
8 :  endfor
9 : return A;
```

### 4.1.2 Recursive Algorithm (Left to Right)

The Left-to-Right algorithm does the opposite of Right-to-Left algorithm. It computes the modular exponentiation by scanning recursively the bits of the private exponent $d$ from the most significant bit (MSB) to least significant bit (LSB). This algorithm requires less memory than the previous one and thus it is considered lighter.

Recursive Algorithm "Left-to-Right modular exponentiation"

INPUT: $m$, $d$, $N$
OUTPUT:$A \equiv m^d mod\, N$

```
1: if (d == 1)
2:   return m;
3: else
4:   // call the function it self with dividing d by 2
5:   A= modularExponentiation(m, b/2, N)
6:   A = A^2 mod N
7:   if(dis odd)
8:     return A = A · m mod N
9:   else
10:      return A = A mod N
```

## 4.2 Modular Exponentiation Attacks

### 4.2.1 Fault Model

Both of our attacks are based on modifying the Public Modulus $N$ during the computation of the Modular Exponentiation. This is done in real life using a laser beam. The laser beam affects $N$ by changing a random byte from $N$ into a random non-zero byte value ($R_8$). For more details see (Sect 5.2).

### 4.2.2 Attack Against "Right-To-Left" Modular Exponentiation

If we assumed that the binary representation of d is $d = \sum_{i=0}^{n-1} 2^i \cdot d_i$. Then, the RSA signature will be written as:

$$S \equiv \dot{m}^{\sum_{i=0}^{n-1} 2^i \cdot d_i} mod N$$

When a fault occurs $j$ steps before the end of the exponentiation, this step will start with a faulty square, regardless of the value of $d_{n-j}$ may be:

$$\hat{B} \equiv (\dot{m}^{2^{n-j-1}} mod N)^2 mod \hat{N}$$

So, becuase of the fault injection the algorithm will keep computing the faulty operations. Thus, the correct operation that executed before the fault injection will be denoted as

$$A \equiv \dot{m}^{\sum_{i=0}^{(n-j-1)} 2^i \cdot d_i} mod N$$

and so we will obtain the following:

$$\hat{S} \equiv ((A \cdot \hat{B})...)\hat{B}^{2^{j-1}} mod \hat{N}$$

$$\equiv A \cdot \hat{B}^{\sum_{i=(n-j)}^{n-1} 2^i \cdot d_i} mod \hat{N}$$

$$\equiv \left[ (\dot{m}^{\sum_{i=0}^{(n-j-1)} 2^i \cdot d_i} mod N) \cdot (\dot{m}^{2^{(n-j-1)}} mod N)^{\sum_{i=(n-j)}^{n-1} 2^{[i-(n-j)+1]} \cdot d_i} \right] mod \hat{N}$$

Since the fault injection has occured during the computation, this results to divide the computation into a correct part and a faulty one. Also, since the attacker can trigger the fault injection using Simple Power Analysis, he knows the values of $j$ for each faulty signature $\hat{S}$. Therefore, he can figure as well the first computation step that has been infected by the fault. As a result, he can figure the number of bits of $d$ that are handled with the wrong modulus.

Now, we are going to give an example of how the algorithm (Algorithm 2 [BCG08]) allows you to retreive the Private Key d. In our example, $l = 1$($l$ is the number of bits you retreive each time). Lets assume that $N = 1141$, $d = 3533$,$m = 9726$ , the $\hat{N} = 11423$(that the laser beam changed $N$into) and the binary representation of $d = 00110111001101_2$.

The first thing we do is obtain a correct signature $S$.

$S = m^d \, mod \, N$
$S = 9726^{3533} \, mod \, 11413 = 5761$

The next step, is to obtain the faulty signatures. Since the first attack starts recovering the bits from the Most Significant Bit (MSB)to the Least Significant Bit (LSB). The first fault location will be 14. But as we saw above the first two bits of $d$are zero(Number of bits of d = 12). So, the first two fault injections will not change $S$and we will not get a fault signature $\hat{S}$.

The first faulty signature $\hat{S}$we will get will be when $jk = 2$. The faulty signature we get will be $\hat{S} = 4994$. Becuase we know that the $d_{11} = 1$, we will start with $d^{'}_{(k)} = 1$ just to show you how the algorithm works. So we start by calcualting $d^{'}$.

$d^{'} := [d^{'}_{(k)} << (n - (j_k + 1))] + d$(We had to change the algorithm to make it work)
$d^{'} := [1 << (14 - (2 + 1))] + 0 = 2048$

We know that the faulty Public Modulus is $\hat{N} = 11423$.
$S^{'}_{(d',\hat{N}')} := [(S \cdot \dot{m}^{-d^{'}}) \, mod \, N \cdot (\dot{m}^{2^{(n-jk-1)}} \, mod \, N)^{2^{[1-(n-jk)]} \cdot d^{'}}] \, mod \, \hat{N}$
$S^{'}_{(d',\hat{N}')} := [(5761 \cdot 9726^{-2048}) \, mod \, 11413 \cdot (9726^{2^{(14-2-1)}} \, mod \, 11413)^{2^{[1-(14-2)]} \cdot 2048}] \, mod \, 11423$
$S^{'}_{(d',\hat{N}')} := [3398 \cdot (4096)^{2^{-11} \cdot 2048}] \, mod \, 11423$
$S^{'}_{(d',\hat{N}')} := [3398 \cdot 4096] \, mod \, 11423 = 4994 = \hat{S}$
After that, we add the recovered bit to d.
$d = d^{'}$

## 4.3 Attack Against "Left-To-Right" Modular Exponentiation

In this part, we apply the same attack principle of the first attack but this time on the "Left-To-Right" Modular Exponentiation. Same as the first attack, the fault is injected into the Public Modulus $N$at step $j_k$, so that the internal register value before the modification is:

$$A \equiv \dot{m}^{\sum_{i=j}^{n-1} 2^{i-j} \cdot d_i} mod \, N$$

And since the perturbed operation is a square, then the faulty signature will be:

$$\begin{aligned} \hat{S} &\equiv (((A^2 . \dot{m}^{d_{j-1}})^2 \cdot \dot{m}^{d_{j-2}})^2 \cdots)^2 \cdot \dot{m}^{d_0} mod \, \hat{N} \\ &\equiv A^{2^j} \cdot \dot{m}^{\sum_{i=0}^{j-1} 2^i . d_i} mod \, \hat{N} \end{aligned}$$

We can see that it splits the equation into two parts, the right part ($A$) and the faulty part after the step $j_k$. Which makes it the same as the first attack, if $\hat{N}$is a prime number, then the square roots can be computed in polynomial time.
The attacker can take advantage of the two parts equation. Since the right internal register $A$ is $j-th$time square multiplied by the message power to the next bit of the private exponent after the $j_k$step. The attacker can find $A^{2^j}$for the searched value $d'_{(k)}$and it can be computed as:

$$R_{(d'_{(k)},\hat{N}_i)} \equiv \hat{S}_k \cdot \dot{m}^{-d'_{(k)}} mod\, \hat{N}_i$$

$\hat{N}_i$have to be a prime number Since $R_{(d'_{(k)},\hat{N})}$is expected to be a multiple quadratic residue ($j_k-th$quadratic residue) modulo $\hat{N}_i$. if its not a quadratic residue then the attacker can deduce that the candidates are wrong and try another ones.
Now for the final modular check. At this point the attacker can find $S'$so that he can compare it to the right signature $S$, which can be found as the following:

$$S' \equiv ((R_{(d'_{(k)},\hat{N}_i)})^{1/2^{j_k}} mod\, \hat{N}_i)^{2^{j_k}} \cdot \dot{m}^{d'_{(k)}} mod\, N$$

Finally, he check if the $S'$satisfies $S$, then he can go and recover the next bit of the private exponent.
To understand the concept more better, we tried an example:
Suppose we have the following values:

$N = 11413 = 14$bits size
$m = 9726$
$d = 3533 = 00110111001101_2$

$$\begin{aligned} S &\equiv m^d mod\, N \equiv (9726)^{3533}\, mod\, 11413 \\ &\equiv 5761\, mod\, 11413 \end{aligned}$$

At $j_k = 3$, the attacker injects a faults into $N$, and lets say the new modified public modulus is $\hat{N} = 11423$which is a prime number. then $S_k \equiv 8356\, mod\, 11423$

As we explained before that equation will split into two parts that will help the attacker to recover $j_k - th$bit of the private exponent $d$if $\hat{N}$is a prime number.

The attacker now is searching for the value of the next bit of the private exponent $d$at $j_k = 3$, and he has already obtained the $d_{recover} = 5$.

we know that the $d_3 = 1$of $d = 3533$. we can see that $d'_{(k)} = d_3 \cdot 2^{j_k} + d_{recover} = 1 \cdot 2^3 + 5 = 13$

Now, we can find $R_{(d'_{(k)}, \hat{N}_i)}$as the following:

$$R_{(d'_{(k)}, \hat{N}_i)} \equiv S_k \cdot m^{-d'_{(k)}} mod\, \hat{N} \equiv 8356 \cdot (9726)^{-13} mod\, 11423 \equiv 8356\, mod\, 11423$$

After that the attacker find the $j_k - th$square roots modulo $\hat{N}$ in order to find the right internal register $A$. So that the attacker check the final modular by finding $S'$

$$\begin{aligned} S' &\equiv ((R_{(d'_{(k)}, \hat{N_i})})^{1/2^{j_k}} mod\, \hat{N}_i)^{2^{j_k}} \cdot \dot{m}^{d'_{(k)}} mod\, N \\ &\equiv ((8356)^{1/2^3} mod\, 11423)^{2^3} \cdot (9726)^{13} mod\, 11413 \\ &\equiv (5440)^{2^3} \cdot (9726)^{13} mod\, 11413 \\ &\equiv 5761\, mod\, 11413 \end{aligned}$$

we can see that $S' \equiv S\, mod\, N \equiv 5761\, mod\, 11413$. And now the attacker adds the recovered bit to d.
$d_{recover} := d'_{(k)}$

### 4.3.1 Tonelli and Shanks' Algorithm

we will use Tonelli and Shanks' algorithm in our attack against "Left-To-Right" algorithm. Tonelli and Shanks' algorithm is an efficient algorithm to compute square roots modulo P, where P is prime number. This algorithm is probabilistic and has a principle that based on isomorphism between the multiplicative group $(\mathbb{Z}/P\mathbb{Z})^*$and the additive group $\mathbb{Z}/(P-1)\mathbb{Z}$. If we assumed that $P - 1$can be written like: $P - 1 = 2^e.r$ and$r$ is odd. Therefore, we can assume that the cyclic group G of order $2^e$ is a subgroup of $\mathbb{Z}/(P-1)\mathbb{Z}$. According to that, we can assume also that $z$ is the generator of G. And so, if we assumed that a is a quadratic residue modulo N, then:

$$a^{(P-1)/2} \equiv (a^r)^{2^{e-1}} \equiv 1 mod\, P$$

Since $a^r mod\, P$ is a square in G, then we can say that there exists an integer k that $k \in [0 : 2^e - 1]$where

$$a^r \cdot z^k = 1$$

in G. Therefore it will be applied to $a^{r+1} \cdot z^k = a$ in G. Note that the square root of a, is represented as

$$a^{1/2} \equiv a^{(r+1)/2} \cdot z^{k/2} mod\, P$$

To summarize, there are two main operation for this algorithm. The first main operation is to find the generator z of the subgroup G. While the other, is to calculate the exponent k.

# 5 Implementation

We started the implementation phase by installing the GIVARO-3.2.13 library. which allows you to use big numbers. Then, we started with building two RSA Signature systems, which we need to implement both of the attacks. The first one using the iterative "Right-To-Left" Modular Exponentiation. The second one using the "Left-To-Right" Modular Exponentiation which runs recursively. After that, we implemented a function that simulates the effects of the the laser beam which is used in real life in both attacks. Moreover, for the Second Attack we had to implement the Tonelli and Shanks Algorithm for Modular Square Root with primes. At the end, we implemented the two attacks algorithms. All these parts will be explained more deeply in this section.

## 5.1 Modular Exponentiation

We implemented both the Modular Exponentiation Algorithms mentioned before, but added the fault injection. In the "Right-To-Left" function we inject the faulty Modulus $\hat{N}$ just before the computation of a multiplication. On the other hand, in the "Left-To-Right" function we change the value of $N$ to $\hat{N}$just before the computation of a square.

| Iterative Algorithm "Right-to-Left modular exponentiation" | Recursive Algorithm "Left-to-Right modular exponentiation" |
|---|---|
| INPUT: $m$, $N$, $d$, $j_k$<br>OUTPUT: $A \equiv m^d mod\, N$ | INPUT: $m$, $N$, $d$, $j_k$<br>OUTPUT: $A \equiv m^d mod\, N$ |
| 1 : $A := 1$;<br>2 : $B := m$;<br>3 : for $i$ from 0 upto $(n-1)$<br>4 : if $(j_k step)$<br>5 : // *inject the fault Modulus* $N = \hat{N}$<br>6: $N = laserBeam(N)$;<br>7 : endif<br>8 : if $(d_i == 1)$<br>9 : $A := (A \cdot B) mod N$ ;<br>10 : endif<br>11 : $B := B^2 mod N$ ;<br>12 : endfor<br>13 : return $A$; | 1: if $(d == 1)$<br>2: return $m$;<br>3: else<br>4: // *call the function it self with dividing d by 2*<br>5: $A$= $modularExponentiation(m, \frac{b}{2}, N)$;<br>6: if($j_k$step)<br>7: // *inject a faults into public modulus* $N = \hat{N}$<br>8: $N = laserBeam(N)$;<br>9: endif<br>10: $A = A^2 \, mod\, N$;<br>11: if($d$is odd)<br>12: return $A = A \cdot m \, mod\, N$;<br>13: else<br>14: return $A = A \, mod\, N$;<br>15: endif<br>16: endif |

## 5.2 Laser Beam

In order to simulate the attacks we needed a function that simulates the effects of the laser beam.

Laser Beam Algorithm

INPUT: $N$
OUTPUT:$\hat{N}$

1 : // $R_8$*represents the random byte value that the laser beam will put.*
1 : $R_8 \in [1; 2^8 - 1]$ ;
2 : // *i represents the random byte the laser beam hits.*
3 : $i \in \left[0; \frac{n}{8} - 1\right]$
4 : $\varepsilon = R_8 \cdot 2^{8_i}$;
5 : $\hat{N} = N \oplus \varepsilon$;
6 : return $\hat{N}$;

## 5.3 Attacks Against Modular Exponentiation

Before implementing the actual attacks there are some values we need to retrieve. First, we need to get a correct signature $S$ using the Modular Exponentiation function. After that, we get the pair set (faulty Signature $\hat{S}$, fault location $j$) which is composed of all the faulty signatures $\hat{S}$ and their fault locations, we use the Modified Modular Exponentiation in this part .

We implemented this part using a for loop. The loop starts from the Most Significant Bit (MSB) and keeps decrementing by $l$ until it reaches the Least Significant Bit (LSB) in the First Attack. However, it does the opposite in the Second Attack, it starts from the LSB and keeps increasing by $l$ upto the MSB. This way the faulty signatures $\hat{S}$will be sorted according to their fault location. In descending order for the First Attack and ascending order for the Second one. the loop puts all the faulty signature $\hat{S}$ it obtains with their fault location $j$ in a set of pairs.

## 5.4 First Attack (Attack Against "Right-To-Left" Modular Exponentiation

After completing the previous steps we implemented the the Algorithm "DFA against "Right-To-Left" Algorithm" which is described in (Perturbating RSA Public Keys: An improved Attack)[Algorithm 2 BCG08] .

## 5.5 Second Attack ( Attack Against "Left-To-Right" Modular Exponentiation)

### 5.5.1 Tonelli and Shanks Algorithm

There are essentially three algorithms to compute the Square Roots Modulo *prime*, one of them is the Tonelli and Shanks algorithm that is quite efficient. That is why we used it in our implementation.[Cohen].

---

Algorithm: Tonelli and Shanks to compute the square roots modulo $p$ (from the book A Cource in Computational Algebraic Number Theory, Author Henri Cohen).

---

INPUT: $a$, $p$
OUTPUT:$x \equiv a^2 mod\, p$

---

1: // $(\frac{a}{p}) = 1$*which mean a quadratic residue*
2: if( $(\frac{a}{p}) \neq 1$)
3: return 0;
4: endif
5: $p - 1 = 2^e \cdot q$, $q$ is odd
6: // *choose numbers* $n$ *until* $(\frac{a}{p}) = -1$
7: while( $(\frac{n_{random}}{p}) \neq -1$ )
8: endwhile
9: $z \equiv n^q \, mod\, p$;
10: $y = z$; $r = e$; $x \equiv a^{(q-1)/2} mod\, p$;
11: $b \equiv a \cdot x^2 \, mod\, p$;
12: $x \equiv a \cdot x \, mod\, p$;
13: while( $b \neq 1$ )
14: // *find the smallest* $m \geq 1$*s.t*
15: for( $m = 1$; $b \neq 1$; $m{+}{+}$)
16: $b \equiv 2^m \, mod\, p$;
17: endfor
18: if( $r == m$ ) // *a is non-quadratic residue modulo p*
19: return $-1$;
20: endif
21: // *reduce the exponent*
22: $t \equiv y^{r-m-1} mod\, p$;
23: $y \equiv t^2 mod\, p$;
24: $r \equiv m \, mod\, p$;
25: $x \equiv x.t \, mod\, p$;
26: $b \equiv b.y \, mod\, p$;
27: endwhile
28: return $x$;

---

### 5.5.2 Dictionary of Prime modulus

In order to implement the second attack algorithm, we need to create a set of all the possible prime values of $\hat{N}$. That is why we implemented the Build Prime Dictionary Algorithm. this step allows the attacker to test all these values with the faulty signatures.

---

Algorithm: Build Prime Dictionary

---

INPUT: $N$, $D_{length}$
OUTPUT: the set of primes $(Dict_i)_{1 \leq i \leq D_{length}}$ ,

---

1: $n =$*the size length of* $N$
2: $i = 1$;
3: // $R_8$is a non zero random byte value
4: for $R_8$from 1upto 255
5: // *pos* is the random byte location in $N$
6: for *pos* from 0upto $(\frac{n}{8}) - 1$
7: $\hat{N} = N \oplus (R_8 \ll 8 \times pos)$;
8: if ( $\hat{N}$is prime )
9: $Dict[i] = \hat{N}$;
10: $i++$;
11: endif
12: endfor
13: endfor
14: return *Dict*;

---

### 5.5.3 The Second Attack Algorithm

After implementing the both the Square Root and the Build_Prime_Dict functions, we implemented the DFA against "Left-To-Right" Algorithm described in (Faults Attacks on RSA Public Keys: Left-To-Right Implementation are also Vulunerable) [Algorithm 3 BCDG09] .

# 6 Results

## 6.1 First Attack Results

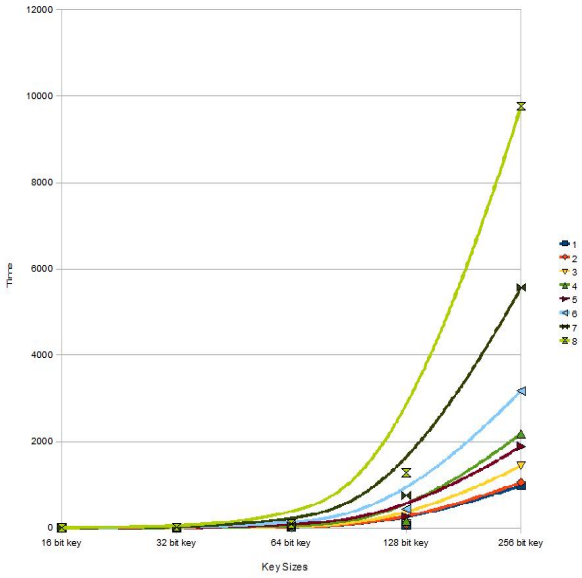


Figure 1: First Attack Results

The tables bellow shows the Total time it takes to run the attack depending on key size and the length of bits recovered each time.

Key Size 16 Bits

| Length | Max No. Iterations | Time for a Single Iteration | Total Time |
|---|---|---|---|
| 1 | 16320 | 0.000015099 | 0.136576 |
| 2 | 16320 | 0.000014541 | 0.172955 |
| 3 | 24480 | 0.000014935 | 0.226950 |
| 4 | 32640 | 0.000014771 | 0.283235 |
| 5 | 65280 | 0.000013207 | 0.416099 |
| 6 | 97920 | 0.000014888 | 0.775602 |
| 7 | 195840 | 0.000008394 | 1.115020 |
| 8 | 261120 | 0.000007831 | 2.221420 |

Key Size 32 Bits

| Length | Max No. Iterations | Time for a Single Iteration | Total Time |
|---|---|---|---|
| 1 | 65280 | 0.000013497 | 0.66 |
| 2 | 65280 | 0.000009926 | 0.69 |
| 3 | 89760 | 0.000013398 | 0.91 |
| 4 | 130560 | 0.000012836 | 1.51 |
| 5 | 228480 | 0.000012871 | 2.8 |
| 6 | 391680 | 0.000013079 | 4.2 |
| 7 | 652800 | 0.000013067 | 8.56 |
| 8 | 1044480 | 0.000012994 | 14.5142 |

Key size 64 Bits

| Length | Max No. Iterations | Time for a Single Iteration | Total Time |
|---|---|---|---|
| 1 | 261120 | 0.000024051 | 5.29 |
| 2 | 261120 | 0.000023726 | 5.67 |
| 3 | 359040 | 0.000023626 | 7.59 |
| 4 | 522240 | 0.000023724 | 12.08 |
| 5 | 848640 | 0.000023656 | 19.57 |
| 6 | 1436160 | 0.000024139 | 33.44 |
| 7 | 2611200 | 0.000023913 | 56.62 |
| 8 | 4177920 | 0.000023910 | 102.32 |

Key Size 128 Bits

| Length | Max No. Iterations | Time for a single Iteration | Total Time |
|---|---|---|---|
| 1 | 1044480 | 0.000076815 | 74.16 |
| 2 | 1044480 | 0.000075328 | 77.8 |
| 3 | 1403520 | 0.000076206 | 106.53 |
| 4 | 2088960 | 0.000074847 | 158.62 |
| 5 | 3394560 | 0.000075361 | 268.83 |
| 6 | 5744640 | 0.000076663 | 439.77 |
| 7 | 9922560 | 0.000075537 | 758.17 |
| 8 | 16711680 | 0.000076229 | 1291.22 |

The max number of iterations feild can also represent the number of the rebuilt signatures $S^{'}$ we have to create in the attack. In our algorithm there are 4 loops that are doing different operations which means each one will takes specific time to run depending on the size of the key and the length of the retreived bits. The first loop starts from 1 and will keep working until it reaches $\lfloor n/l \rfloor$. We can calculate the number of times this loop run by calculating L1 as the following:

$$L1 = Ksize/l$$

where Ksize is the key size and $l$ is the length of the retrieved bits. The second loop will start running from 0 until $2^l - 1$which means it will run for $2^l$ times.

The third loop will start runnig from 1 until $\left(2^8 - 1\right)$and thus the maximum number of times it will run is 255 times.

The forth loop will start runing from 0 up to $(Ksize/8 - 1)$. So, the number of times this loop will be running is $Ksize/8$.

So, depending on what previously explained, the maximum number of iterations can be calculated as the following:

$$MaxNo = L1 \cdot 2^l \cdot 255 \cdot Ksize/8$$

## 6.2 Second Attack Results

From the test we run on the second attack we obtained the following results. The test has been tested with $l = 1$and $l = 2$. The results shows that the $l = 2$takes less time than when we run the second attack with $l = 1$. We could not successfly implement the attack for $3 \leq l \leq 8$ because we didn't have enough time.

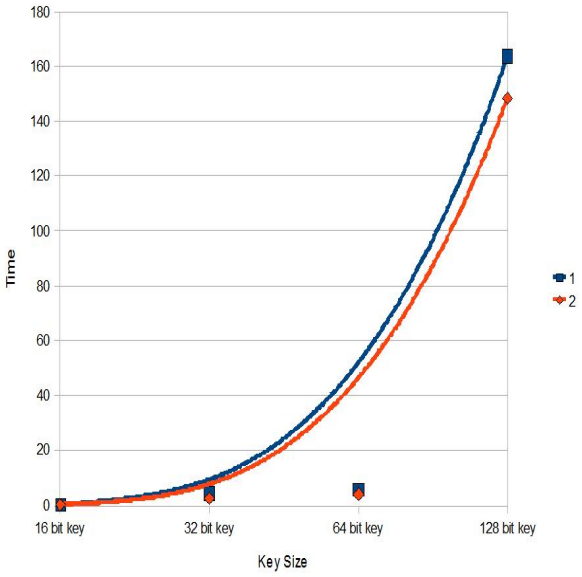


Figure 2: Second Attack Results

### 6.3 Square Root Results

After testing the square root algorithm. We got different results that are described in the following figure:

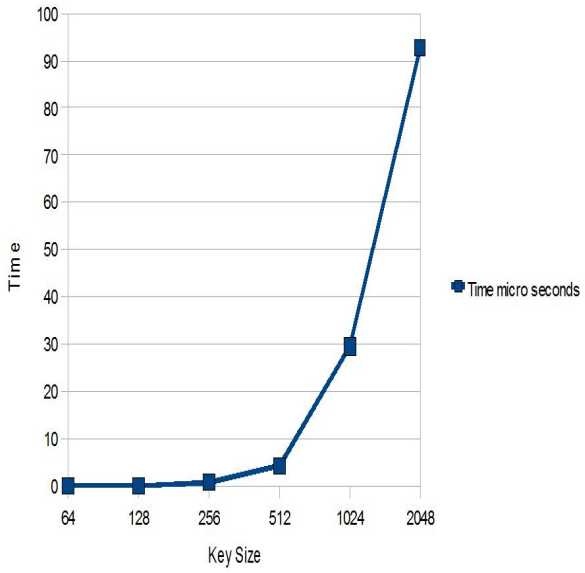


Figure 3: Square Root Algorithm Results

## 7 Conclusion

Finally, we managed to obtain detailed results for the First Attack with different $l$ sizes. And we we able to successfully recover the Private Key $d$with $n$=256 bits and that is due to the fact that it takes a very long time to test the attack on bigger $n$. On the other hand, for the Second Attack we were only able to successfully recover the Private Key $d$with $l$=1and $l$=2 with $n$=128 bits. From our results, we concluded that the Second Attack is faster than the First one but requires more attempts, because doesn't work unless all the faulty Public Modulus $\hat{N}$are prime. Moreover, we found out that every time you increase the $l$ it takes more time to recover the Private Key $d$. But, we assume that with bigger key sizes increasing the $l$ might decrease the time of the attack.

## References


[BCG08] A. Berzati, C. Canovas, and L. Goubin. Perturbating RSA Public Keys: an Improved Attack. In Cryptography Hardware and Embedded Systems (CHES 2008), Lecture Notes in Computer Science. Springer-Verlag, 2008.

[BCDG09] A. Berzati, C. Canovas, J. Dumas and L. Goubin. Fault Attacks on RSA Public Keys: Left-To-Right Implementation are also Vulnerable. (2009)

[Cohen] Henry Cohen. A Course in Computational Algebraic Number Theory. Springer (1996)