Seifert's RSA Fault Attack: Simplified Analysis and Generalizations

James A. Muir*
School of Computer Science
Carleton University
jamuir@scs.carleton.ca

15 December 2005 21:11:36 EST

Abstract

Seifert recently described a new fault attack against an implementation of RSA signature verification. Here we give a simplified analysis of Seifert's attack and gauge its practicality against RSA moduli of practical sizes. We suggest an improvement to Seifert's attack which has the following consequences: if an adversary is able to cause random faults in only 4 bits of a 1024-bit RSA modulus stored in a device, then there is a greater than 50% chance that they will be able to make that device accept a signature on a message of their choice. For 2048-bit RSA, 6 bits suffice.

1 Introduction

Recently, Seifert described a novel attack against an implementation of the RSA signature verification operation [5]. His attack is based on the following assumptions:

- An adversary has a device which contains an RSA public key, (N, e), stored in protected read-only memory (e.g. in EEPROM).
- The values N and e are known to the adversary.
- On input (m, s), the device transfers the values N and e from protected memory and proceeds to check if s is a valid signature for m.
- As the device transfers the value N from protected memory, *the adversary can induce data faults*.

The attacker's goal is to create a message-signature pair which the device will accept as valid. Seifert describes a probabilistic algorithm which does this. Moreover, Seifert's attack is a *selective forgery*; that is, an adversary is able to select a message, compute a "signature" on it and have the device accept these as a valid message-signature pair. This is all done without factoring N and without computing the private key, d.

Seifert's attack uses an incredibly simple strategy: if forging RSA signatures using the modulus N is too difficult, then modify some bits of N and create a new modulus, $\widehat{N}$, where it *is* easy to forge signatures. Seifert points out that it is very easy to create signatures when $\widehat{N}$ is prime, since then we can simply compute the private exponent, $\widehat{d}$, as $e^{-1} \bmod (\widehat{N}-1)$, assuming that e is relatively prime to $\widehat{N}-1$. In the off-line part of Seifert's attack, the adversary can *choose* which bits of N to modify to create $\widehat{N}$. In the on-line part of

*J.A. Muir is supported by a Natural Sciences and Engineering Research Council Postdoctoral Fellowship.

the attack, the adversary repeatedly queries the device with a specially constructed message-signature pair and causes data faults until this particular $\widehat{N}$ is used as the modulus in the signature validation algorithm.

To put a practical perspective on Seifert's attack, imagine that the device is a "locked" computer that will only execute code if it can validate a signature on that code. This is exactly what Microsoft had hoped to implement in its Xbox game-console [7]. Microsoft attempted to design the Xbox so that only software signed by Microsoft would run on it. However, a number of Xbox enthusiasts found ways to circumvent Microsoft's software authentication techniques [3]. In fact, Seifert credits Andy Green and Franz Lehner's Xbox "hack" (see [3], p. 143) as the inspiration for his attack. However, there is an important distinction between the two techniques. Green and Lehner's attack involves a *deterministic* change to an internal parameter; Seifert's attack involves a *random* change to an internal parameter. If an attacker has the ability to change bits of (N, e) deterministically, then it is much easier to unlock the device. In this case, it is possible to defeat the authentication procedure by just setting e to equal 1.

In this paper, we present a modification of Seifert's attack. We demonstrate that we do not need to restrict ourselves to errors only in the least significant bits of the modulus. Also, we show that we do not need to limit ourselves to moduli, $\widehat{N}$, that are prime; what we are really after is moduli that have easily computed factorizations. We give a simplified analysis for our attack and we compare this analysis against some computational trials using RSA public keys of practical sizes (i.e. 1024 bits and 2048 bits).

Outline In §2 we describe the fault model which use throughout the paper. In §3 we review Seifert's attack and adapt it to our fault model. An analysis and some computational results are presented in §3.1 and §3.2. In §4 we give an improvement to Seifert's attack. Analysis, computational results and an example are provided in §4.1, §4.2 and §4.3. We end with some remarks in §5.

2 Fault Model

Suppose the device implements the following RSA signature verification algorithm:

FAULTY-RSA-VERIFY(m, s)

comment: the device's embedded RSA public key is (N, e).

$(\widehat{N}, \widehat{e}) \leftsquigarrow (N, e)$
$h \leftarrow H(m)$
$h' \leftarrow s^{\widehat{e}} \bmod \widehat{N}$
if $h = h'$
  then return "accept"
  else return "reject"

The operator "$\leftsquigarrow$" denotes an assignment operation that is subject to random bit faults. We will make this more precise in a moment. The function H denotes a message encoding function which typically incorporates some cryptographic hash function. For example, H might be a full-domain hash function constructed from a concatenation of SHA-256 hashes.

The bit faults which affect the public key are instigated by the adversary. In our model, *we only consider bit faults in the RSA modulus, N*. These faults change the value N to $\widehat{N}$. This is a *non-deterministic* process and so we can consider $\widehat{N}$ to be a random variable. We assume that the public exponent e is unaffected by bit faults. It seems unlikely that an adversary would be able to take advantage of errors in e. However, if by randomly flipping bits of e, we could obtain a value $\widehat{e}$ for which it is easy to compute $\widehat{e}$-th roots modulo N, then this type of attack would certainly be worth exploring. But, this seems to happen very rarely (unless the adversary has a way to set $\widehat{e} = 1$ with high probability). In practice, e is usually taken to be 3 or 65537 since these values help make signature verification more efficient.

The effect of faults on the modulus can be described using an *error function*, ξ. This function takes two parameters; the first is N and the second is a random variable, Δ. Both ξ and Δ determine how N is transformed. The definition of ξ is highly dependent on the architecture of the device and the technique that the adversary uses to cause faults.

One possible definition of ξ is the following

$$\xi(N, \Delta) = N \oplus 0^{n-b-c} \| \Delta \| 0^c, \quad \text{where} \quad \Delta \in \{0, 1\}^b. \tag{1}$$

Here, N is considered as an n-bit array; its value is changed by xoring it with a b-bit string, Δ, which is offset according to the value c. The values b and c are fixed. This error function models *random-data fixed-location* faults (i.e. random data appears at a fixed location within the modulus).

Another possible definition of ξ is this:

$$\xi(N, \Delta) = N \odot 1^{n-b-\Delta} \| 0^b \| 1^{\Delta}, \quad \text{where} \quad \Delta \in \{0, 1, 2, \ldots n - b\}. \tag{2}$$

The symbol $\odot$ denotes a bit-wise "and" of two bit-strings. Now, the bits of N are changed by zeroing a block of b bits which is offset according to the parameter Δ (which is now an integer). This error function models *fixed-data random-location* faults (i.e. constant data appears at a random location within the modulus).

In general, we can consider

$$\xi : \{0, 1\}^n \times S \rightarrow \{0, 1\}^n$$

where S is a finite set. The random variable Δ is drawn from S. In the algorithm Faulty-RSA-Verify, after the operation $(\widehat{N}, \widehat{e}) \leftsquigarrow (N, e)$, we have that

$$\widehat{N} = \xi(N, \Delta), \text{ for some } \Delta \in S, \quad \text{and} \quad e = \widehat{e}.$$

The number of possible values of $\widehat{N}$ is related to the size of S. When Faulty-RSA-Verify executes, the adversary initiates faults but they *cannot* control the value of Δ.

An excellent survey of techniques for inducing computational faults in a device is presented in [1]. For example, a random-data fixed-location fault can be induced by illuminating one of the device's registers or data buses with a strong light source. Alternately, a fixed-data random-location fault can be initiated by varying the device's supply voltage.

For the sake of clarity, we continue our exposition assuming that ξ is defined as in (1). Thus, we have

$$\widehat{N} = N \oplus 0^{n-b-c} \| \Delta \| 0^c, \text{ for some } \Delta \in \{0, 1\}^b.$$

We sometimes refer to Δ as an *error vector*. Δ is b-bits wide and this might be influenced by the size of the device's data-bus or registers; for example, many smart card have 8-bit registers while typical desktop PCs have 32-bit registers. The bit-length of the modulus is n, so we have $n = \lfloor \lg N \rfloor + 1$.

Using the parameters, b, c, Δ, we can rewrite the algorithm Faulty-RSA-Verify like so:

FAULTY-RSA-VERIFY(m, s)

comment: the device's embedded RSA public key is (N, e). n is the bit-length of N.
b is the length of the error vector and c is its offset.

$\Delta \in_R \{0, 1\}^b$
$\widehat{N} \leftarrow N \oplus 0^{n-b-c} \| \Delta \| 0^c$
$\widehat{e} \leftarrow e$

$h \leftarrow H(m)$
$h' \leftarrow s^{\widehat{e}} \bmod \widehat{N}$
if $h = h'$
then return "accept"
else return "reject"

3 Seifert's Attack

Here is a simplified description of Seifert's attack which we have adapted according to our fault model:

RSA-ATTACK$(m, (N, e))$

comment: m is a message selected by the adversary.

$S \leftarrow \{0, 1\}^b \setminus \{0^b\}$
repeat
$\Delta \in_R S$
$S \leftarrow S \setminus \{\Delta\}$
$\widehat{N} \leftarrow N \oplus 0^{n-b-c} \| \Delta \| 0^c$
until ($\widehat{N}$ is prime **and** $\gcd(e, \widehat{N} - 1) = 1$) **or** ($S = \varnothing$)

if $S = \varnothing$
then return "fail"

$\widehat{d} \leftarrow e^{-1} \bmod (\widehat{N} - 1)$
$h \leftarrow H(m)$
$s \leftarrow h^{\widehat{d}} \bmod \widehat{N}$

repeat
output $\leftarrow$ FAULTY-RSA-VERIFY(m, s)
until *output* = "accept"

return "success"

Essentially, what is happening here is that we randomly flip bits of N until we find a value $\widehat{N}$ such that $\widehat{N}$ is prime and e^{-1} exists modulo $\widehat{N} - 1$. If we find such a value, then we use it to construct a new private exponent $\widehat{d}$ by computing the inverse of e modulo $\widehat{N} - 1$. This can be done efficiently using the extended Euclidean algorithm or Fermat's Theorem. Next, we generate a signature for m, using $\widehat{d}$, which will verify against the public key $(\widehat{N}, e)$. All of this work so far is done *off-line* (i.e. it does not require us to interact with the device). The attack finishes with an *on-line* phase where we repeatedly query the device with our selected message and the signature we constructed for it. Each time we query the device, we hope that the bit faults we initiate will cause the device to use the modulus $\widehat{N}$ when it checks our message and signature.

When Seifert presented his attack he did not consider the parameter c. The bit faults he considered were always restricted to the b least significant bits of N (i.e. when $c = 0$). Our model is more general. We will see that the value of c has no effect on the running time or success probability of the attack. The value of b does, however.

3.1 Analysis

The procedure RSA-ATTACK contains two iterative loops. The first loop is executed during the off-line portion of the attack:

$S \leftarrow \{0,1\}^b \setminus \{0^b\}$
repeat
 $\Delta \in_R S$
 $S \leftarrow S \setminus \{\Delta\}$
 $\widehat{N} \leftarrow N \oplus 0^{n-b-c} \| \Delta \| 0^c$
until ($\widehat{N}$ is prime **and** $\gcd(e, \widehat{N} - 1) = 1$) **or** ($S = \varnothing$)

Note that the error space S may be traversed in other ways. Instead of selecting error vectors uniformly at random from S, it might be more convenient to consider them in lexicographic order (i.e. counting from 1 to $2^b - 1$ in binary).

The off-line portion of the attack succeeds if we can find a value of $\widehat{N}$ that causes the loop to exit before we exhaust the error space. The probability of this happening for a particular value of $\widehat{N}$ is

$$\Pr(\widehat{N} \text{ is prime}) \cdot \Pr(\gcd(e, \widehat{N} - 1) = 1).$$

In practice, e is usually equal to 3 or 65537 which are both prime numbers. We will make the simplifying assumption that e is prime. Thus,

$$\Pr(\gcd(e, \widehat{N} - 1) = 1) = \Pr(e \nmid \widehat{N} - 1) = \frac{e-1}{e}.$$

A consequence of the Prime Number Theorem is that the probability that a random *odd* positive integer x is prime is roughly $2/\ln x$. Using this fact, and the bound $2^{n-1} \le \widehat{N} < 2^n$, we have

$$\Pr(\widehat{N} \text{ is prime}) \approx \frac{2}{\ln \widehat{N}} > \frac{2}{\ln 2^n} = \frac{2}{n \ln 2}.$$

The reader who carefully examines the definition of RSA-ATTACK may notice that there are some values of $\widehat{N}$ that are *not* necessarily odd. This happens only when $c = 0$. However, in the off-line phase of the attack, since we are searching for $\widehat{N}$ that are prime, when carrying out our search we would simply modify the error space, S, so that $\widehat{N}$ is always odd. When $c > 0$, no modification is necessary. When $c = 0$, we would set S equal to $\{0,1\}^{b-1}\{0\} \setminus \{0^b\}$ instead of $\{0,1\}^b \setminus \{0^b\}$.

Now we can estimate the probability that $\widehat{N}$ meets our criteria as

$$\frac{2(e-1)}{e \cdot n \ln 2}.$$

So, we expect that we will have to consider about $\frac{e \cdot n \ln 2}{2(e-1)}$ values of $\widehat{N}$ before we find one that suits our needs. The probability that there is *no* good value of $\widehat{N}$ inside our search space can be estimated as

$$\left(1 - \frac{2(e-1)}{e \cdot n \ln 2}\right)^{2^b - 1}.$$

off-line stage	worst case running time expected running time probability of success	$O(2^b - 1)$ $O\left(\frac{e \cdot n \ln 2}{2(e-1)}\right)$ $1 - \left(1 - \frac{2(e-1)}{en \ln 2}\right)^{2^b - 1}$
on-line stage	expected running time	$O(2^b)$

FIGURE 1: Characteristics of RSA-ATTACK.

This represents the probability that the off-line stage of the attack *fails*.

The *on-line* portion of the attack is described in the second iterative loop:

repeat
  output ← FAULTY-RSA-VERIFY(m, s)
until *output* = "accept".

This portion of the attack is much simpler to analysis. We want the RSA verification algorithm to be affected by a particular error vector. Assuming that each error vector from $\{0, 1\}^b$ is equiprobable, this happens with probability $\frac{1}{2^b}$. So, we expect to have to carry out about 2^b faulted signature verification operations before the desired error occurs.

Some of the important characteristics of RSA-ATTACK are summarized in Figure 1. Notice how the parameter b affects the success probability and running time of the attack. By increasing b we can increase the probability that the off-line stage of the attack succeeds. However, this also increases the expected number of steps in the on-line stage of the attack. Depending on how quickly the target device processes and responds to on-line queries, the expected number of on-line queries required can present a major obstacle to attack implementors.

3.2 The off-line search in practice

We constructed two RSA public keys by pairing the RSA challenge numbers RSA-1024 and RSA-2048 [9] with the exponent $e = 65537$. For each public key, we examined the search space used in the off-line stage of RSA-ATTACK for various values of b and c (recall that b is the length of the error vector and c is its offset). All our numerical computations (i.e. probabilistic primality testing and gcd's) were done using the C++ library NTL [6].

For each public key, we took $b \in \{4, 6, 8, 10, 12, 14, 16\}$. For each value of b, we set c to equal each multiple of b in the interval $0 \ldots n - b - 1$; so, c takes on $1 + \left\lfloor \frac{n-b-1}{b} \right\rfloor$ different values. In theory, the offset, c, could take any value in the interval $0 \ldots n - b$; our reason for limiting c to multiples of b was that we wanted the c values to define disjoint search spaces.

We illustrate our experiments with an example. Suppose $b = 4$ and $n = 1024$. For these parameters, the error offset c takes on 255 different values; namely, $0, 4, 8, 12, \ldots, 1016$. Each value of c defines a search space which is disjoint from all the others. We found that 3 of the 255 search spaces contained $\widehat{N}$ values for which the off-line stage of the attack succeeds. The ratio $3/255$ can be compared to our estimate of the probability that the off-line stage of the attack succeeds when $b = 4$ (see below). Across the 255 search spaces, we examined $255 \cdot (2^4 - 1) = 3825$ values of $\widehat{N}$. Of these 3825 values, 3 had the desired properties. The ratio $3/3825$ can be compared to our estimate of the probability that $\widehat{N}$ is prime and $\widehat{N} - 1$ is relatively prime to $e = 65537$. The same methodology was used for the other values of b. Our experimental results are summarized in Figure 2.

	b	good $\widehat{N}$'s	total # of $\widehat{N}$'s	ratio	good c's	total # of c's	ratio
RSA-1024 $e = 65537$	4	3	3825	0.00078	3	255	0.0118
	6	24	10710	0.00224	23	170	0.135
	8	68	32385	0.00210	53	127	0.417
	10	264	104346	0.00253	97	102	0.951
	12	969	348075	0.00278	85	85	1
	14	3354	1195959	0.00280	73	73	1
	16	11658	4128705	0.00282	63	63	1
RSA-2048 $e = 65537$	4	11	7665	0.00144	11	511	0.0215
	6	44	21483	0.00205	41	341	0.120
	8	106	65025	0.00163	80	255	0.314
	10	332	208692	0.00159	164	204	0.804
	12	1018	696150	0.00146	169	170	0.994
	14	3433	2391918	0.00144	146	146	1
	16	11601	8322945	0.00139	127	127	1

FIGURE 2: Experimental results for the off-line stage of RSA-ATTACK.

From our analysis in the previous section, for the 1024-bit public key, we estimate the probability that a value of $\widehat{N}$ has the desired properties as

$$\frac{2 \cdot 65536}{65537 \cdot 1024 \cdot \ln 2} \approx 0.00282.$$

The empirical values listed for RSA-1024 in column 5 of Figure 2 appear to converge toward this estimate. Thus, we expect to have to examine about $1/0.00282 = 355$ values of $\widehat{N}$ before we find one that meets our criteria. If the architecture of a device permits the attacker some control over the size of b, then they might choose b so that their search space contains at least 355 values (but, of course, this does not guarantee that the search space will contain a good value of $\widehat{N}$). In practice, it would seem prudent to first find lots of good values of $\widehat{N}$, for various values of b and c, and then pick one that has a short error vector which is easy to instantiate in the device.

Using the probability above, we can estimate the probability that the off-line stage of the attack will succeed for different values of b:

$$\begin{aligned} b &= 4, & 1-(1-0.00282)^{2^4-1} &\approx 0.0415 \\ b &= 6, & 1-(1-0.00282)^{2^6-1} &\approx 0.163 \\ b &= 8, & 1-(1-0.00282)^{2^8-1} &\approx 0.513 \\ b &= 10, & 1-(1-0.00282)^{2^{10}-1} &\approx 0.944. \end{aligned}$$

These estimates are quite close to the empirical values listed for RSA-1024 in column 8 of Figure 2.

Similar comparisons can be made for RSA-2048. We estimate the probability that a 2048-bit value of $\widehat{N}$ has the desired properties as

$$\frac{2 \cdot 65536}{65537 \cdot 2048 \cdot \ln 2} \approx 0.00141.$$

And, we estimate the probability that the off-line stage of the attack will succeed for different values of b as:

$$\begin{aligned}
b = 4, &\quad 1 - (1 - 0.00141)^{2^4-1} \approx 0.0209 \\
b = 6, &\quad 1 - (1 - 0.00141)^{2^6-1} \approx 0.0851 \\
b = 8, &\quad 1 - (1 - 0.00141)^{2^8-1} \approx 0.302 \\
b = 10, &\quad 1 - (1 - 0.00141)^{2^{10}-1} \approx 0.764.
\end{aligned}$$

These estimates are quite close to our empirical results.

Although our fault model greatly simplifies many of the arguments Seifert worked through in his paper, from these experiments it appears that our analysis does give an accurate picture of what can be expected in practice.

4 Improving Seifert's Attack

The criteria that Seifert uses for his off-line search can be relaxed. When we examine various values of $\widehat{N}$, what we really want is an integer that has an *easily computed prime factorization*. If $\widehat{N}$ is prime, then this is certainly true. However, there are many other integers which have this property. If we know the prime factorization of $\widehat{N}$, then we can easily compute $\varphi(\widehat{N})$ and then use the extended Euclidean algorithm to obtain $\widehat{d} = e^{-1} \mod \varphi(\widehat{N})$.

Deciding whether or not the prime factorization of a random integer can be easily computed is a subjective task. It depends upon what factorization method you are using, how efficiently is it implemented and how much time you are willing to invest. The strategy we used was this: given $\widehat{N}$, divide out any prime factors $\le 2^{10}$, and then check whether the quotient is equal to 1 or is prime. We chose a small bound of 2^{10} since we did not want to invest much time in attempting to factor each $\widehat{N}$. There is a convenient data structure in NTL which can be used to generate all the primes less than 2^{30} in sequence[1].

Using this approach, the off-line stage of the attack becomes:

$S \leftarrow \{0, 1\}^b \setminus \{0^b\}$
repeat
  $\Delta \in_R S$
  $S \leftarrow S \setminus \{\Delta\}$
  $\widehat{N} \leftarrow N \oplus 0^{n-b-c} \| \Delta \| 0^c$
  $\widehat{N}_0 \leftarrow \widehat{N}$ with any prime factors $\le 2^{10}$ divided out.
until ($\widehat{N}_0$ is prime or equal to 1 **and** $\gcd(e, \varphi(\widehat{N})) = 1$) **or** ($S = \varnothing$)

Obviously, the bound 2^{10} can be replaced with one larger or smaller according to the preference of the implementor.

4.1 Analysis

The probability that a value of $\widehat{N}$ causes the loop above to exit is

$$\begin{aligned}
\Pr(\widehat{N}_0 \text{ is prime or equal to } 1) \cdot \Pr(\gcd(e, \varphi(\widehat{N})) = 1) &= \\
\Pr(\text{the second-largest prime factor of } \widehat{N} \text{ is } \le 2^{10}) \cdot \Pr(\gcd(e, \varphi(\widehat{N})) = 1).&
\end{aligned}$$

[1] Actually, during our experiments, we found that the largest prime generated by NTL's `PrimeSeq` class to be $2^{30} - 2^{16} - 1$ which is not the greatest prime $\le 2^{30}$. There 3184 more primes which are larger.

		Seifert's off-line search		*off-line search using primes* $\leq 2^{10}$		*off-line search using primes* $\leq 2^{30}$	
	b	expected iterations	probability of success	expected iterations	probability of success	expected iterations	probability of success
RSA-1024 $e = 65537$	4	355	0.041	57	0.233	19	0.564
	6	355	0.084	57	0.421	19	0.820
	8	355	0.513	57	0.989	19	0.999
RSA-2048 $e = 65537$	4	710	0.021	115	0.123	38	0.331
	6	710	0.043	115	0.238	38	0.564
	8	710	0.302	115	0.893	38	0.999

FIGURE 3: Comparison of off-line search strategies.

The distribution of the second-largest prime factor of random integers $\leq x$ as $x \to \infty$ was investigated by Knuth and Trabb Pardo [4]. Following their discussion, we define

$$F_2(\beta) := \lim_{x\to\infty} \Pr(\text{a random integer } \leq x \text{ has its second-largest prime factor } \leq x^{\beta}).$$

This limit was shown to exist. Over the interval $0 \leq \beta \leq 1/2$, $F_2(\beta)$ increases monotonically from 0 to 1; for $\beta \geq 1/2$, $F_2(\beta) = 1$. Since n is the bit-length of the modulus, N, we have $\widehat{N} \leq 2^n$. Setting x and β equal to 2^n and $10/n$, respectively, we obtain

$$F_2\,(10/n) \approx \Pr(\text{a random integer } \leq 2^n \text{ has its second-largest prime factor } \leq 2^{10}).$$

Assuming $\widehat{N}$ behaves like a random integer $\leq 2^n$, this is the probability that we want to approximate. Using our assumption that e is prime, we estimate the probability that $\widehat{N}$ meets our criteria as

$$\frac{(e-1)F_2(10/n)}{e}.$$

Unfortunately, $F_2(\beta)$ does not have a simple closed form so it is not immediate what sort of improvement this achieves. However, we can quantify the difference by plugging in some numbers.

Evaluating $F_2(\beta)$ requires some careful work and, fortunately, a table of values is provided in [4]. From this table, we use Lagrange interpolation to build a polynomial approximation to $F_2(\beta)$ in the interval $0 \leq \beta \leq 1/2$. This gives us

$$F_2(10/1024) \approx 0.0175, \quad F_2(10/2048) \approx 0.00872,$$
$$F_2(30/1024) \approx 0.0538, \quad F_2(30/2048) \approx 0.0264.$$

Now, for a 1024-bit public key with $e = 65537$, the probability that a random value of $\widehat{N}$ ends our search when we cast out primes $\leq 2^{10}$ is roughly

$$\frac{65536 \cdot 0.0175}{65537} \approx 0.0175.$$

So, our chances, which we calculated in §2.4, have increased from 0.282% to 1.75%. If we cast out primes less than 2^{30}, we get 5.38%. Some more comparisons are made in Figure 3. The most dramatic difference appears in the number of values of $\widehat{N}$ we expect to consider before the search ends.

	b	good $\widehat{N}$'s	total # of $\widehat{N}$'s	ratio	good c's	total # of c's	ratio
RSA-1024	4	63	3825	0.0165	54	255	0.212
$e = 65537$	6	191	10710	0.0178	111	170	0.653
	8	545	32385	0.0168	126	127	0.992
	10	1843	104346	0.0177	102	102	1
	12	6018	348075	0.0173	85	85	1
	14	20861	1195959	0.0174	73	73	1
	16	72711	4128705	0.0176	63	63	1
RSA-2048	4	63	7665	0.00822	60	511	0.117
$e = 65537$	6	203	21483	0.00945	155	341	0.455
	8	598	65025	0.00920	228	255	0.894
	10	1863	208692	0.00893	204	204	1
	12	6259	696150	0.00899	170	170	1
	14	20910	2391918	0.00874	146	146	1
	16	72968	8322945	0.00877	127	127	1

FIGURE 4: Experimental results of searching for easily factorable $\widehat{N}$'s

4.2 The improved off-line search in practice

We repeated the experiments from §2.4 using our new on-line search criteria. For various error widths and offsets, we exhausted the resulting search spaces and determined which $\widehat{N}$'s could be easily factorized after casting out primes $\leq 2^{10}$. Our results are summarized in Figure 4. Our empirical results are close to what our analysis predicts.

4.3 An Example

The 2048-bit modulus from the public RSA key that Microsoft stores inside the Xbox can be found in publicly available source code [8]. Here is the modulus in hexadecimal:

```
A44B1BBD7EDA72C7143CD5C2D4BA880C7681832D5198F75FCAB1618598E2B3E4
8D9A47B0BFF6BC967CAE88F198266E535A6CB41B470C0A38A19D8F57CB11F568
DB52CF69E49F604EEA52F4EB9D37E80C60BD70A5CF5A67EC05AA6B3E8C80C116
819A14892BFA7603BECE39F09C42724EE9F371C473AAA09FEDA34F9EA1019827
BD07CA52A80013BE9471E46FCF1CA4D915FB9DF95E9344330B6AAE0B90526AD1
BE475D10797526075C9206FF758A3EB3BAF7C0A22E51645BB9F13FE129A22F2E
1BEDDA95D68AFC6D46585B01FBB5737273C6AEE399148C5B8E77B479DE8B05BD
EEC27FEFFF7B349C64F51002D2F6522ED43617F2A1A3D4C2E6D73D66E54ED7D3
```

Some of the techniques for inducing faults explained in [1] can be used to zeroize bytes of data. It is interesting to consider an off-line search, with respect to the modulus above, in this fault model.

The modulus above consists of 256 bytes. If we index the bytes from least significant (byte 0) to most significant (byte 255), then the smallest index, i, such that when we zeroize byte i we obtain an easily factorable number is $i = 16$. The method of factorization we used was to cast out all prime factors $\leq 2^{30}$ and then apply a probabilistic primality test. The factorization is $3 \cdot 13 \cdot 199 \cdot 856469 \cdot p_0$ where p_0 is a large prime.

The smallest index, j, such that when we zeroize byte j we obtain a prime number is $j = 104$.

5 Remarks

Our analysis and computational trials show that if an adversary is able to cause random faults in *only 4 bits* of a 1024-bit RSA modulus stored in a device, then there is a greater than 50% chance that they will be able to make that device accept a signature on a message of their choice. For 2048-bit RSA, *6 bits* suffice. This is an improvement over Seifert's original attack which required 8 and 10 bits, respectively.

These percentages do not take into account any of the practical difficulties that might be involved in a real-world implementation of the attack. For example, it might be difficult to limit the effect of faults to a particular block of bits within the modulus. Our examination was limited to a mathematical model and so we did not deal with these issues. Presently, there is no record of anyone successfully carrying out this attack in the open literature. But, when the technical requirements for the attack are compared to what is reported in [1], its practicality does not seem that far fetched.

One way to defend against this attack is to have the device check the integrity of its public key. This might be done by computing a cryptographic hash of the public key and then comparing it to some stored value. However, care must be taken in when this comparison is done. If an integrity check is done before a signature is verified, this will not stop attackers who cause bit faults in the public key after the check. Other countermeasures, against fault analysis attacks in general, are discussed in [1].

An interesting lesson that can be taken from Seifert's attack is that RSA public keys are somewhat fragile; that is, if you flip a few bits of an RSA modulus then there is a non-negligible probability that you end up with an integer that is easy to factor. It would be interesting to know if other public key cryptosystems not based on the integer factorization problem exhibit this behaviour. The discrete log problem, as it is posed in DSA, does not seem to have this susceptibility (i.e. if $y = g^x$ is a DSA public key, then, by flipping bits of y, it does not seem likely we will obtain a value $\widehat{y}$ for which it is easy to compute $\log_g \widehat{y}$; in fact, the probability that $\widehat{y}$ is in the subgroup generated by g is negligible.).

References

[1] H. BAR-EL, H. CHOUKRI, D. NACCACHE, M. TUNSTALL AND C. WHELAN. The sorcerer's apprentice guide to fault attacks. *Cryptology ePrint Archive*, Report 2004/100.

[2] D. BONEH, R. DEMILLO, AND R. LIPTON. On the importance of checking cryptographic protocols for faults. *Journal of Cryptology* **14** (2001), 101–119.

[3] A. HUANG. *Hacking the Xbox: An Introduction to Reverse Engineering*, No Starch Press, 2003.

[4] D. KNUTH AND L. TRABB PARDO Analysis of a simple factorization algorithm. *Theoretical Computer Science* **3** (1976), 321–348.

[5] J. SEIFERT. On authenticated computing and RSA-based authentication. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 2005)*, November 2005, pp. 122–127.

[6] V. SHOUP. NTL: A Library for doing Number Theory (version 5.4). `http://shoup.net/ntl/`

[7] Microsoft Xbox, `http://www.microsoft.com/xbox/`

[8] Operation X, `http://sourceforge.net/projects/opx/`

[9] RSA Challenge Numbers, `http://www.rsasecurity.com/rsalabs/node.asp?id=2093`