Seifert's RSA Fault Attack: Simplified Analysis and Generalizations

James A. Muir[*]

School of Computer Science
Carleton University, Ottawa, Canada
`http://www.scs.carleton.ca/~jamuir`

Abstract. Seifert (ACM CCS 2005) recently described a new fault attack against an implementation of RSA signature *verification*. Seifert's attack differs from the seminal work of Boneh, DeMillo and Lipton (EUROCRYPT 1997) in that it targets a public-key rather than a private-key operation. Here we give a simplified analysis of Seifert's attack and gauge its practicality against RSA moduli of practical sizes. Our intent is to give practice-oriented work estimates rather than asymptotic results. We also 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.

Keywords: hardware faults, fault analysis, signature verification, RSA signatures.

1 Introduction

Recently, Seifert described a novel attack against an implementation of the RSA signature verification operation [8]. 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 to working memory, and then 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

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

an arbitrary 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 learning 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 modifies some of the least significant bits of N to create $\widehat{N}$. In the on-line part of 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 verification 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 [10]. 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 [5]. In fact, Seifert credits Andy Green and Franz Lehner's Xbox "hack" [5, page 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.

After the publication of Seifert's attack, one of the most pressing questions concerning it involved its practical consequences (e.g., What is the estimated work factor and success probability for an adversary who mounts the attack against a 1024-bit public RSA key?). The analysis provided in [8] gives a number of asymptotic expressions for the work factor and success probability of the attack, but extracting practical information from them is nontrivial. For example, the main result (Thm. 1) in [8] says that if an adversary can cause faults in the least significant $O(\lg\lg N)$ bits of N, then, assuming that the Riemann Hypothesis is true, they can make the device accept a signature on an arbitrary message with probability $\Theta(1/\lg N)$ in $\lg^{O(1)} N$ time. Clearly, the real-world implications of this result for 1024-bit RSA depend on the constants hidden in the asymptotic terms. It turns out that it is possible to give a more precise and straightforward analysis of Seifert's attack. With this analysis it is possible to make statements like the following: If an adversary can cause faults in only 4 bits of an RSA modulus, then there is a greater than 50% chance that they will be able to make the device accept a signature on an arbitrary message after 2^4 on-line queries.

Our contributions. Our main contribution is a simplified analysis of Seifert's attack. The work estimates we present are practice-oriented and can be easily interpreted. We verify our analysis against some computational trials which use RSA public keys of practical sizes (i.e., 1024 bits and 2048 bits). In addition, we offer two straightforward generalizations to 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 really want is moduli that have easily computed factorizations.

Outline. In §2 we describe the fault model which we 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 and computational results are provided in §4.1 and §4.2. We briefly discuss some open problems related to fault attacks on discrete log based signature schemes in §5. We end with some remarks in §6.

2 Fault Model

Suppose that the target device (i.e., the locked computer) implements Algorithm 1. In this algorithm, the operator "$\leftsquigarrow$" denotes an assignment operation

Algorithm 1. Faulty RSA Signature Verification

```
Input: $m \in \{0,1\}^*$, $s \in \mathbb{Z}_N$.
Output: "accept" or "reject".
1: $(\widehat{N}, \widehat{e}) \leftsquigarrow (N, e)$
2: $h \leftarrow H(m)$
3: $h' \leftarrow s^{\widehat{e}} \bmod \widehat{N}$
4: if $h = h'$ then return "accept"
5: else return "reject"
```

that is subject to 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 [2].

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 N to $\widehat{N}$ non-deterministically while e remains unchanged. This assumption – that faults can be localized to a particular parameter of a cryptographic computation – is commonly used in the theory of fault analysis (cf. [3]).

Recently, Naccache, Nguyen, Tunstall and Whelan [7] presented a key recovery attack on DSA which requires that an adversary zeroize some of the least-significant bits of the nonce k used in signature generation. What's more, they successfully implemented their attack against a smartcard and demonstrated that it is possible for an adversary to engineer the required data faults. However, despite the physical experiments conducted by Naccache et al., the practicality of the assumption that bit-faults will affect N but not e depends upon the characteristics of a specific target device and the tools and skills of a specific adversary (the fact that the bit-length of e is much shorter than N may be of some help). So let us state plainly that the validity of our assumptions for any particular target device are untested; however, experience shows that often attacks can be modified minorly to adapt to different situations, and thus we believe this fault model is important to consider as a starting point. For example, if an adversary can localize faults to N only 20% of the time, this may suffice to carry out the attack.

Concerning bit-faults in e, it seems unlikely that an adversary would be able to take advantage of such errors. But, 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. However, unless the adversary has a way to set $\widehat{e} = 1$ with high probability, this would seem to happen very rarely. In practice, e is usually taken to be 3 or 65537 since these values help make signature verification more efficient.

An excellent survey of techniques for inducing computational faults in a device is presented in [1]. For example, a *random-data fixed-location* fault (i.e., random data appears at a fixed location within N) 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 (i.e., constant data appears at a random location within the modulus) can be initiated by varying the device's supply voltage.

We model the effect of faults on the modulus using an *error function*, ξ. This function takes two parameters: the first is N and the second is a nonce, Δ. Both ξ and Δ determine how N is transformed. One possible definition of ξ is the following

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

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 non-negative integers that satisfy $0 \le b + c \le n$. This error function models random-data fixed-location faults.

Another possible definition of ξ is

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

The symbol "&" denotes a bit-wise "and". Now, the bits of N are changed by zeroing a block of b bits offset according to the parameter Δ (which is now an integer). This error function models fixed-data random-location faults.

In general, we can consider

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

where S is a finite set. The nonce Δ is drawn uniformly from S (we denote this as $\Delta \in_R S$). In Algorithm 1, after the operation $(\widehat{N}, \widehat{e}) \leftsquigarrow (N, e)$, we have that

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

When Algorithm 1 is executed, the adversary initiates faults but they *cannot* control the value of $\mathit{\Delta}$.

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

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

$\mathit{\Delta}$ is b-bits wide; we sometimes refer to $\mathit{\Delta}$ as an *error vector*. The value of b might be influenced by the size of the device's data-bus or registers; for example, many smart cards 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 $b, c, \mathit{\Delta}$, we can rewrite the signature verification operation like so:

Algorithm 2. Faulty RSA Signature Verification

Input: $m \in \{0,1\}^*$, $s \in \mathbb{Z}_N$.
Output: "accept" or "reject".

```
1: $\mathit{\Delta} \in_R \{0,1\}^b$
2: $\widehat{N} \leftarrow N \oplus 0^{n-b-c} \| \mathit{\Delta} \| 0^c$
3: $\widehat{e} \leftarrow e$
4: $h \leftarrow H(m)$
5: $h' \leftarrow s^{\widehat{e}} \bmod \widehat{N}$
6: if $h = h'$ then return "accept"
7: else return "reject"
```

3 Seifert's Attack

In Algorithm 3 we present a simplified description of Seifert's attack which is adapted according to our fault model. Note that the title "Algorithm" is applied loosely. To turn the description into a true algorithm, we would need to bound the number of times that the second iterative loop (lines 11-13) is executed.

Essentially, what is happening in Algorithm 3 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 any interaction 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.

Algorithm 3. Seifert's Fault Attack

Input: An arbitrary message $m \in \{0,1\}^*$, the device's RSA public key (N, e).
Output: "success" or "fail".

```
 1: $S \leftarrow \{0,1\}^b \setminus \{0^b\}$
 2: repeat
 3:     $\Delta \in_R S$
 4:     $S \leftarrow S \setminus \{\Delta\}$
 5:     $\widehat{N} \leftarrow N \oplus 0^{n-b-c} \| \Delta \| 0^c$
 6: until ($\widehat{N}$ is prime and $\gcd(e, \widehat{N}-1) = 1$) or ($S = \varnothing$)
 7: if $S = \varnothing$ then return "fail"
 8: $\widehat{d} \leftarrow e^{-1} \bmod (\widehat{N}-1)$
 9: $h \leftarrow H(m)$
10: $s \leftarrow h^{\widehat{d}} \bmod \widehat{N}$
11: repeat
12:     output ← the output of Algorithm 2 on input $m, s$.
13: until output = "accept"
14: return "success"
```

Note that Algorithm 3 generalizes Seifert's original attack model in an obvious way. The original model did not consider the parameter c as bit-faults were always restricted to the b least-significant bits of N. We will see that the value of c has no effect on the running time or success probability of the attack; however, the value of b does.

3.1 Analysis

Algorithm 3 contains two iterative loops. The first loop (lines 2-6) is executed during the off-line portion of the attack:

```
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 $S = \{0,1\}^b \setminus \{0^b\}$ may be traversed in other ways (e.g., it might be more convenient to enumerate the elements of S in lexicographic order).

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 $2^{n-1} \leq \widehat{N} < 2^n$, 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 Algorithm 3 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.

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

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

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

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

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 (lines 11-13):

```
repeat
    output ← the output of Algorithm 2 on input $m, s, (N, e)$.
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 $\frac{1}{2^b}$. Thus, the expected number of faulted signature verification operations needed before the desired error occurs is 2^b.

Some of the important characteristics of Algorithm 3 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	worst case running time	$O(2^b - 1)$
	expected running time	$O\left(\frac{e \cdot n \ln 2}{2(e-1)}\right)$
	probability of success	$1 - \left(1 - \frac{2(e-1)}{en \ln 2}\right)^{2^b - 1}$
on-line stage	expected running time	$O(2^b)$

Fig. 1. Characteristics of Algorithm 3

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 [12] with the exponent $e = 65537$. For each public key, we examined the search space used in the off-line stage of Algorithm 3 for various values of b and c (recall that b is the error-width and c is its offset). All our numerical computations (i.e., probabilistic primality testing and gcd's) were done using the C++ library NTL [9].

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 $0 \dots n - b - 1$; so, c takes on $1 + \lfloor \frac{n-b-1}{b} \rfloor$ different values. In theory, the offset, c, could take any value in the interval $0 \dots n - b$; 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, $0, 4, 8, 12, \dots, 1016$. 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.

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. From the probability above, we see that the expected number of values of $\widehat{N}$ we must examine before we find one that meets our criteria is $1/0.00282 = 355$. 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-width which is easy to instantiate in the device.

	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

Fig. 2. Experimental results for the off-line stage of Algorithm 3

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:

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

These estimates are quite close to our empirical results.

Although our fault model and method of analysis greatly simplify many of the arguments from Seifert's paper, these experiments demonstrate 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 it is implemented and how much time you are willing to invest. The strategy we used was this: given $\widehat{N}$, divide out any prime factors $\leq 2^{10}$, 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 $\widehat{N}$ in our simulations. Adversaries who are willing to invest more time into attempting to factorize a few values of $\widehat{N}$ might utilize, say, the elliptic curve factoring method since it tends to find small prime factors of $\widehat{N}$ first.

The off-line stage of the attack now 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 $\leq 2^{10}$ 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. There is a convenient data structure in NTL which can be used to generate all primes less than 2^{30} in sequence[1].

4.1 Analysis

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

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

The distribution of the second-largest prime factor of random integers $\leq x$ as $x \to \infty$ was investigated by Knuth and Trabb Pardo [6]. Following their discussion, we define

[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 $\leq 2^{30}$; there are 3184 more primes which are larger.

$$F_2(\beta) := \lim_{x\to\infty} \Pr(\text{a random integer} \le x \text{ has its 2nd-largest prime factor} \le x^\beta).$$

Knuth and Trabb Pardo showed that this limit exists, and also presented a method for approximating its value. Over the interval $0 \le \beta \le 1/2$, $F_2(\beta)$ increases monotonically from 0 to 1; for $\beta \ge 1/2$, $F_2(\beta) = 1$. Since n is the bit-length of the modulus, N, we have $\widehat{N} \le 2^n$. Setting x and β equal to 2^n and $10/n$, respectively, we obtain

$$F_2\left(10/n\right) \approx \Pr(\text{a random integer} \le 2^n \text{ has its 2nd-largest prime factor} \le 2^{10}).$$

Assuming $\widehat{N}$ behaves like a random integer $\le 2^n$, 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.

A table of values for $F_2(\beta)$ is provided in [6]. From this table, we build a polynomial approximation to $F_2(\beta)$ in the interval $0 \le \beta \le 1/2$. 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 prime factors $\le 2^{10}$ 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.

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 prime factors $\le 2^{10}$. Our results are summarized in Figure 4. Our empirical results are close to what our analysis predicts.

Although we have considered only random-data fixed-location faults in our experiments, it is just as easy to treat fixed-data random-location faults. An interesting demonstration of this is presented in Appendix A.

		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	4	355	0.041	57	0.233	19	0.564
$e = 65537$	6	355	0.084	57	0.421	19	0.820
	8	355	0.513	57	0.989	19	0.999
RSA-2048	4	710	0.021	115	0.123	38	0.331
$e = 65537$	6	710	0.043	115	0.238	38	0.564
	8	710	0.302	115	0.893	38	0.999

Fig. 3. Comparison of off-line search strategies

	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

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

5 Further Work

An interesting lesson that can be taken from Seifert's attack is that public-key authentication systems based on the integer factorization problem are somewhat fragile with respect to bit-faults; that is, if you randomly flip a few bits of the public-key then there is a non-negligible probability that you end up with an integer that is easy to factor. It would be interesting to determine if similar fault attacks exist for discrete log based authentication systems. We briefly consider some problems related to the ElGamal signature scheme [4].

In the ElGamal signature scheme

– the *private key* is $x \in_R [1, p-2]$.
– the *public key* is (y, g, p) where $y = g^x \mod p$, p is a large prime (say, 1024 bits) and g is a generator of $\mathbb{Z}_p{}^*$.

The signature verification algorithm is presented in Algorithm 4.

Algorithm 4. ElGamal Signature Verification

Input: $m, (r, s), (y, g, p)$
Output: "accept" or "reject"

```
1: if $r \notin [0, p-1]$ or $s \notin [0, p-2]$ then
2:         return "reject"
3: $h \leftarrow H(m)$
4: $u \leftarrow g^h \bmod p$
5: $u' \leftarrow r^s y^r \bmod p$
6: if $u = u'$ then return "accept"
7: else return "reject"
```

To forge a signature on a message m, we must find (r, s) such that

$$g^{H(m)} \equiv r^s y^r \pmod{p}. \tag{3}$$

It is well-known that the forgery problem is no harder than computing discrete logs in $\mathbb{Z}_p^*$ (to see this, choose r at random and then solve a discrete log to obtain s). Now, if we are able to randomly flip bits in one of p, y, g, it may be that the forgery problem becomes tractable.

For example, if we change p to $\widehat{p}$, where $\widehat{p}$ is easily factorable, then we can apply the CRT to $\mathbb{Z}_{\widehat{p}}$ which may reduce our work in solving $g^{H(m)} \equiv r^s y^r \pmod{\widehat{p}}$ using, say, an index-calculus method. Also, if $\varphi(\widehat{p})$ is smooth, then the Pohlig-Hellman algorithm can be utilized; however, we ideally want an attack that has a high probability of success.

One tempting possibility to consider is replacing g with $\widehat{g}$ where the order of $\widehat{g}$ is small (as in a small subgroup attack). However, the probability that randomly flipping bits of g moves us into a small subgroup of $\mathbb{Z}_p^*$ seems to be negligible.

It is possible that computing $\log_g \widehat{y}$ might be easier than computing $\log_g y$. This motivates the following relaxation of the discrete log problem: given y, find $\widehat{y}$ and $\widehat{x}$ such that $\widehat{y} = g^{\widehat{x}}$ and $y \oplus \widehat{y}$ has low weight.

6 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.

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 or unsuccessfully carrying out this attack in the open literature. Whether this is because the assumptions of the attack are too strong or that simply no one has yet attempted to implement it remains to be seen.

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].

Acknowledgements

The author thanks Paul Van Oorschot for providing helpful comments on an earlier version of this paper.

References

1. H. Bar-El, H. Choukri, D. Naccache, M. Tunstall and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE* **94** (2006), 370–382.
2. M. Bellare and P. Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. In "Advances in Cryptology – Eurocrypt '96", *Lecture Notes in Computer Science* **1070** (1996), 399–416.
3. D. Boneh, R. DeMillo, and R. Lipton. On the importance of checking cryptographic protocols for faults. *Journal of Cryptology* **14** (2001), 101–119.
4. T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory* **31** (1985), 469–472.
5. A. Huang. *Hacking the Xbox: An Introduction to Reverse Engineering*, No Starch Press, 2003.
6. D. Knuth and L. Trabb Pardo. Analysis of a simple factorization algorithm. *Theoretical Computer Science* **3** (1976), 321–348.
7. D. Naccache, P. Nguyen, M. Tunstall, C. Whelan. Experimenting with faults, lattices and the DSA. In "Public Key Cryptography – PKC 2005", *Lecture Notes in Computer Science* **3386** (2005), 16–28.
8. 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.
9. V. Shoup. NTL: A library for doing number theory (version 5.4). `http://shoup.net/ntl/`
10. Microsoft Xbox, `http://www.microsoft.com/xbox/`
11. Operation X, `http://sourceforge.net/projects/opx/`
12. RSA Challenge Numbers, `http://www.rsasecurity.com/rsalabs/node.asp?id=2093`

A Fixed-Data Faults

Some of the techniques for inducing faults explained in [1] can be used to zeroize bytes of data. An off-line search for an easily factorable modulus with respect to this fault model is easy carry out. Here, we present an interesting example of this.

The 2048-bit modulus from the public RSA key that Microsoft stores inside the Xbox can be found in publicly available source code [11]. Here is the modulus in hexadecimal:

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

This modulus consists of 256 bytes. If we index the bytes from least significant (byte $0 = \texttt{D3}$) to most significant (byte $255 = \texttt{A4}$), 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 $j = 104$.