Experimenting with Faults, Lattices and the DSA

David Naccache[1,2], Phong Q. Nguyễn[3], Michael Tunstall[2,4], and Claire Whelan[5]

[1] Gemplus Card International, Applied Research & Security Centre,
34 rue Guynemer, Issy-les-Moulineaux, F-92447, France.
david.naccache@gemplus.com[†]

[2] Royal Holloway, University of London, Information Security Group,
Egham, Surrey TW20 0EX, UK.
david.naccache@rhul.ac.uk[†]

[3] CNRS/École normale supérieure, Département d'Informatique,
45 rue d'Ulm, F-75230 Paris Cedex 05, France.
http://www.di.ens.fr/~pnguyen, Phong.Nguyen@di.ens.fr[†]

[4] Gemplus Card International, Applied Research & Security Centre,
Avenue des Jujubiers, La Ciotat, F-13705, France.
michael.tunstall@gemplus.com

[5] School of Computing, Dublin City University,
Ballymun, Dublin 9, Ireland.
cwhelan@computing.dcu.ie[††]

Abstract. We present an attack on DSA smart-cards which combines physical fault injection and lattice reduction techniques. This seems to be the first (publicly reported) physical experiment allowing to concretely pull-out DSA keys out of smart-cards. We employ a particular type of fault attack known as a *glitch attack*, which will be used to actively modify the DSA nonce k used for generating the signature: k will be tampered with so that a number of its least significant bytes will flip to zero. Then we apply well-known lattice attacks on El Gamal-type signatures which can recover the private key, given sufficiently many signatures such that a few bits of each corresponding k are known. In practice, when one byte of each k is zeroed, 27 signatures are sufficient to disclose the private key. The more bytes of k we can reset, the fewer signatures will be required. This paper presents the theory, methodology and results of the attack as well as possible countermeasures.

Keywords: DSA, fault injection, glitch attacks, lattice reduction.

[†] The work described in this paper has been supported in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT. The information in this document reflects only the authors' views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

[††] Supported by the Irish Research Council for Science, Engineering and Technology (IRCSET)

1 Introduction

Over the past few years fault attacks on electronic chips have been investigated and developed. The theory developed was used to challenge public key cryptosystems [4] and symmetric ciphers in both block [3] and stream [8] modes.

The discovery of fault attacks (1970s) was accidental. It was noticed that elements naturally present in packaging material of semiconductors produced radioactive particles which in turn caused errors in chips [11]. These elements, while only present in extremely minute parts (two or three parts per million), were sufficient to affect the chips' behaviour, create a charge in sensitive silicon areas and, as a result, cause bits to flip. Since then various mechanisms for fault creation and propagation have been discovered and researched. Diverse research organisations such as the aerospace industry and the security community have endeavoured to develop different types of fault injection techniques and devise corresponding preventative methods. Some of the most popular fault injection techniques include variations in supply voltage, clock frequency, temperature or the use of white light, X-ray and ion beams.

The objectives of all these techniques is generally the same: corrupt the chip's behaviour. The outcomes have been categorised into two main groups based on the long term effect that the fault produced. These are known as *permanent* and *transient* faults. Permanent faults, created by purposely inflicted defects to the chip's structure, have a permanent effect. Once inflicted, such destructions will affect the chip's behavior permanently. In a transient fault, silicon is locally ionized so as to induce a current that, when strong enough, is falsely interpreted by the circuit as an internal signal. As ionization ceases so does the induced current (and the resulting faulty signal) and the chip recovers its normal behavior.

Preventive measures come in the form of software and hardware protections (the most cost-effective solution being usually a combination of both). Current research is also looking into fault detection where, at stages through the execution of the algorithm, checks are performed to see whether a fault has been induced [10]. For a survey of the different types of fault injection techniques and the various software and hardware countermeasures that exist, we refer the reader to [2].

In this paper we will focus on a type of fault attack known as a glitch attack. Glitch attacks use transient faults where the attacker deliberately generates a voltage spike that causes one or more flip-flops to transition into a wrong state. Targets for insertion of such 'glitches' are generally machine instructions or data values transferred between registers and memory. Results can include the replacement of critical machine instructions by almost arbitrary ones or the corruption of data values.

The strategy presented in this paper is the following: we will use a glitch to reset some of the bytes of the nonce k, used during the generation of DSA signatures. As the attack ceases, the system will remain fully functional. Then, we will use classical lattice reduction techniques to extract the private signature key from the resulting glitched signatures (which can pass the usual verification process). Such lattice attacks (introduced by Howgrave-Graham and Smart [9], and improved by Nguyễn and Shparlinski [14]) assume that a few bits of k are known for sufficiently many signatures, without addressing how these bits could be obtained. In [14], it was reported that in practice, the lattice attack required as few as three bits of k, provided that about a hundred of such signatures were available. Surprisingly, to the authors' knowledge, no fault attack had previously exploited those powerful lattice attacks.

The paper is organised as follows: In section 2 we will give a brief description of DSA, we will also introduce the notations used throughout this paper. An overview of the attack's physical and mathematical parts will be given in section 3. In section 4 we will present the results of our attack while countermeasures will be given in section 5.

Related work: In [1] an attack against DSA is presented by Bao *et al.*, this attack is radically different from the one presented in this paper and no physical implementation results are given. This attack was extended in [6] by Dottax. In [7], Knudsen and Giraud introduce another fault attack on the DSA. Their attack requires around 2300 signatures (*i.e.* 100 times more than the attack presented here). The merits of the present work are thus twofold: we present a new (*i.e.* unrelated to [7, 1, 6]) efficient attack and describe what is, to the authors' best knowledge, the first (publicly reported) physical experiment allowing to concretely pull-out DSA keys out of smart-cards. The present work shows that the hypotheses made in the lattice attacks [9, 14] can be realistic in certain environments.

2 Background

In this section we will give a brief description of the DSA.

2.1 DSA Signature and Verification

The system parameters for DSA [12] are $\{p, q, g\}$, where p is prime (at least 512 bits), q is a 160-bit prime dividing $p-1$ and $g \in \mathbb{Z}_p^*$ has order q. The private key is an integer $\alpha \in \mathbb{Z}_q^*$ and the public key is the group element $\beta = g^\alpha \pmod p$.

Signature: To sign a message m, the signer picks a random $k < q$ and computes:

$$r \leftarrow \left(g^k \pmod p\right) \pmod q \quad \text{and} \quad s \leftarrow \frac{\text{SHA}(m) + \alpha r}{k} \pmod q$$

The signature of m is the pair: $\{r, s\}$.

Verification: To check $\{r, s\}$ the verifier ascertains that:

$$r \stackrel{?}{=} \left(g^{wh}\beta^{wr} \pmod p\right) \pmod q \text{ where } w \leftarrow \frac{1}{s} \pmod q \text{ and } h \leftarrow \text{SHA}(m)$$

3 Attack Overview

The attack on DSA proceeds as follows: we first generate several DSA signatures where the random value generated for k has been modified so that a few of k's least[6] significant bytes are reset[7]. This faulty k will then be used by the card to generate a (valid) DSA signature. Using lattice reduction, the secret key α can be recovered from a collection of such signatures (see [14, 9]). In this section we will detail each of these stages in turn, showing first how we tamper with k in a closed environment and then how we apply this technique to a complete implementation.

3.1 Experimental Conditions

DSA was implemented on a chip known to be vulnerable to V_{cc} glitches. For testing purposes (closed environment) we used a separate implementation for the generation of k.

A 160-bit nonce is generated and compared to q. If $k \geq q - 1$ the nonce is discarded and a new k is generated. This is done in order to ascertain that k is drawn uniformly in $\mathbb{Z}_q^*$ (assuming that the source used for generating the nonce is perfect). We present the code fragment (modified for simplicity) that we used to generate k:

```
PutModulusInCopro(PrimeQ);
RandomGeneratorStart();

status = 0;
do {
  IOpeak();
  for (i=0; i<PrimeQ[0]; i++) {
    acCoproMessage[i+1] = ReadRandomByte();
  }
  IOpeak();

  acCoproMessage[0] = PrimeQ[0];
  LoadDataToCopro(acCoproMessage);

  status = 1;
  for (j=0; j<(PrimeQ[0]+1); j++) {
    if (acCoproResult[j] != acCoproMessage[j]) {
      status = 0;
```

[6] It is also possible to run a similar attack by changing the most significant bytes of k. This is determined by the implementation.

[7] It would have also been possible to run a similar attack if these bytes were set to FF.

```
      }
    }
  }
  while (status == 0);
  RandomGeneratorStop();
```

Note that `IOpeaks`[8], featured in the above code was also included in the implementation of DSA. The purpose of this is to be able to easily identify the code sections in which a fault can be injected to produce the desired effect. This could have been done by monitoring power consumption but would have greatly increased the complexity of the task.

The tools used to create the glitches can be seen in figure 1 and figure 2. Figure 1 is a modified CLIO reader which is a specialised high precision reader that allows one glitch to be introduced following any arbitrarily chosen number of clock cycles after the command sent to the card. Figure 2 shows the experimental set up of the CLIO reader with the oscilloscope used during our experiments. A BNC connector is present on the CLIO reader which allows the I/O to be easily read; another connector produces a signal when a glitch is applied (in this case used as a trigger). Current is measured using a differential probe situated on top of the CLIO reader.

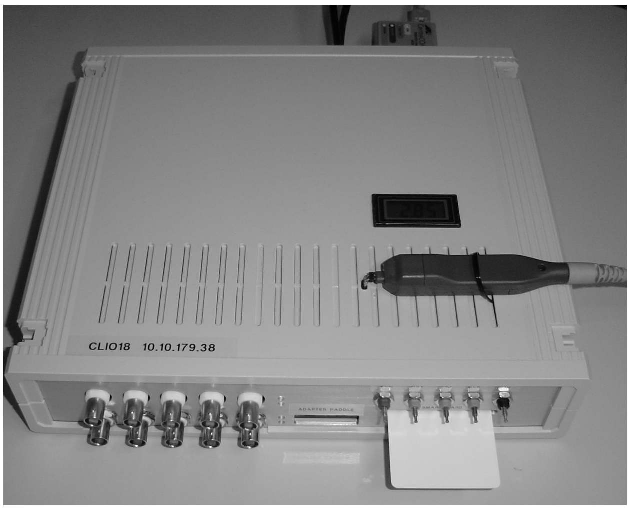

Fig. 1. A Modified CLIO Reader

[8] The I/O peak is a quick movement on the I/O from one to zero and back again. This is visible on an oscilloscope but is ignored by the card reader.

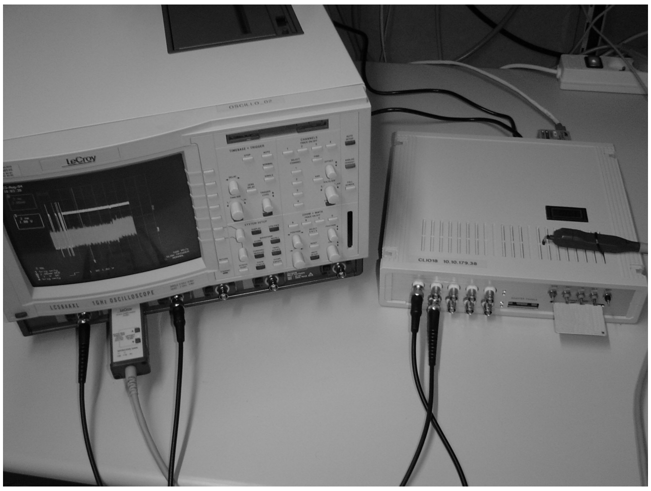

Fig. 2. Experimental Set Up

3.2 Generating a Faulty k

The command that generated k was attacked in every position between the two `IOpeaks` in the code. It was found that the fault did not affect the assignment of k to the RAM *i.e.* the instruction `acCoproMessage[i+1] = ReadRandomByte();` which always executed correctly. However, it was possible to change the evaluation of `i` during the loop. This enabled us to select the number of least significant bytes to be reset. In theory, this would produce the desired fault in k with probability $q/2^{160}$, as if the modified k happens to be larger than q, it is discarded anyway. In practice this probability is likely to be lower as it is unusual for a fault to work correctly every time.

An evaluation of a position that resetted the last two bytes was performed. Out of 2000 attempts 857 were corrupted. This is significantly less than what one would expect, as the theoretical probability is $\simeq 0.77$. We expected the practical results to perform worse than theory due to a slight variation in the amount of time that the smart card takes to arrive at the position where the data corruption is performed. There are other positions in the same area that return k values with the same fault, but not as often.

3.3 The Attack: Glitching k During DSA Computations

The position found was equated to the generation of k in the command that generates the DSA signature. This was done by using the last I/O event at the end of the command sent as a reference point and gave a rough position of where the fault needs to be injected.

As changes in the value of k were not visible in the signature, results would only be usable with a certain probability. This made the attack more complex, as the subset signatures having faulty k values had to be guessed amongst those acquired by exhaustive search.

To be able to identify the correct signatures the I/O and the current consumption signals were monitored during the attacks. An example of such a monitoring is given in figure 3. The object of these acquisitions was to measure the time T elapsed between the end of the command sent to the card and the beginning of the calculation of r. This can be seen in the current consumption, as the chip will require more energy when the crypto-coprocessor is ignited. If we denote

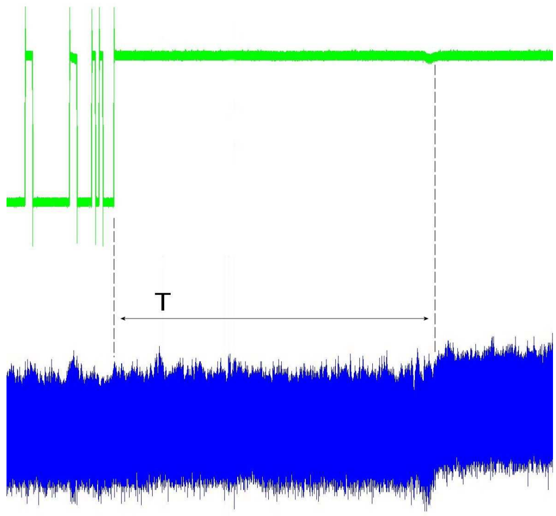

Fig. 3. I/O and Current Consumption (Beginning of the Trace of the Command Used to Generate Signatures).

by t the time that it takes to reach the start of the calculation of r knowing that the picked k was smaller that q (*i.e.* that it was not necessary to restart the picking process) then, if $T = t$ we know that the command has executed properly and that k was picked correctly the first time. If $T > t$ then any fault targeting k would be a miss (as k was regenerated given that the value of k originally produced was greater than q). Signatures resulting from commands that feature such running times can be discarded as the value of k will not present any exploitable weaknesses. When $T < t$ we know that the execution of the code generating k has been cut short, so some of the least significant bytes will be equal to zero. This allows signatures generated from corrupted k values to be identified *a posteriori*.

As the position where the fault should be injected was only approximately identified, glitches were injected in twenty different positions until a position that produced signatures with the correct characteristics (as described above) was found. The I/O peaks left in the code were used to confirm these results. Once the correct position identified, more attacks were conducted at this position to acquire a handful of signatures. From a total of 200 acquisitions 38 signatures where $T < t$ were extracted.

This interpretation had to be done by a combination of the I/O and the current consumption, as after the initial calculation involving k the command no longer takes the same amount of time. This is because $0 < k \leq q$ and therefore k does not have a fixed size; consequently any calculations k is involved in will not always take the same amount of time.

3.4 Use of Lattice Reduction to Retrieve α

We are now in a position to apply the well-known lattice attacks of [9, 14] on El Gamal-type signature schemes: given many DSA signatures for which a few bits of the corresponding k are known, such attacks recover the DSA signer's private key. In our case, these known bits are in fact 0 bits, but that does not matter for the lattice attack. We recall how the lattice attacks work, using the presentation of Nguyễn and Shparlinski [14]. Roughly speaking, lattice attacks focus on the linear part of DSA, that is, they exploit the congruence $s \leftarrow \frac{\mathrm{SHA}(m)+\alpha r}{k} \pmod q$ used in the signature generation, not the other congruence $r \leftarrow \left(g^k \pmod p\right) \pmod q$ which is related to a discrete log problem. When no information on k is available, the congruence reveals nothing, but if partial information is available, each congruence discloses something about the private key α: by collecting sufficiently many signatures, there will be enough information to recover α. If ℓ bits of k are known for a certain number of signatures, we expect that about $160/\ell$ signatures will suffice to recover α. Here is a detailed description of the attack.

For a rational number z and $m \geq 1$ we denote by $\lfloor z \rfloor_m$ the unique integer a, $0 \leq a \leq m-1$ such that $a \equiv z \pmod m$ (provided that the denominator of z is relatively prime to m). The symbol $|.|_q$ is defined as $|z|_q = \min_{b \in Z} |z - bq|$ for any real z.

Assume that we know the ℓ least significant bits of a nonce $k \in \{0, \ldots, q-1\}$ which will be used to generate a DSA signature (for the case of other bits, like most significant bits or bits in the middle, see [14]).

That is, we are given an integer a such that $0 \leq a \leq 2^\ell - 1$ and $k - a = 2^\ell b$ for some integer $b \geq 0$. Given a message m (whose SHA hash is h) signed with the nonce k, the congruence

$$\alpha r \equiv sk - h \pmod q,$$

can be rewritten for $s \neq 0$ as:

$$\alpha r 2^{-\ell} s^{-1} \equiv (a - s^{-1}h)2^{-\ell} + b \pmod q. \tag{1}$$

Now define the following two elements

$$t = \left\lfloor 2^{-\ell} r s^{-1} \right\rfloor_q,$$
$$u = \left\lfloor 2^{-\ell}(a - s^{-1}h) \right\rfloor_q$$

and remark that both t and u can easily be computed by the attacker from the publicly known information. Recalling that $0 \leq b \leq q/2^\ell$, we obtain

$$0 \leq \lfloor \alpha t - u \rfloor_q < q/2^\ell.$$

And therefore:

$$|\alpha t - u - q/2^{\ell+1}|_q \leq q/2^{\ell+1}. \tag{2}$$

Thus, the attacker knows an integer t and a rational number $v = u + q/2^{\ell+1}$ such that :

$$|\alpha t - v|_q \leq q/2^{\ell+1}.$$

In some sense, we know an approximation of αt modulo q. Now, suppose we can repeat this for many signatures, that is, we know d DSA signatures $\{r_i, s_i\}$ of hashes h_i (where $1 \leq i \leq d$) such that we know the ℓ least significant bits of the corresponding nonce k_i. From the previous reasoning, the attacker can compute integers t_i and rational numbers v_i such that :

$$|\alpha t_i - v_i|_q \leq q/2^{\ell+1}.$$

The goal of the attacker is to recover the DSA private key α. This problem is very similar to the so-called hidden number problem introduced by Boneh and Venkatesan in [5]. In [5, 14], the problem is solved by transforming it into a lattice closest vector problem (for background on lattice theory and its applications to cryptography, we refer the reader to the survey [16]; a similar technique was recently used in [13]).

More precisely, consider the $(d+1)$-dimensional lattice L spanned by the rows of the following matrix:

$$\begin{pmatrix} q & 0 & \cdots & 0 & 0 \\ 0 & q & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \ldots & 0 & q & 0 \\ t_1 & \ldots & \ldots & t_d & 1/2^{\ell+1} \end{pmatrix}. \tag{3}$$

The inequality $|v_i - \alpha t_i|_q \leq q/2^{\ell+1}$ implies the existence of an integer c_i such that:

$$|v_i - \alpha t_i - q c_i| \leq q/2^{\ell+1}. \tag{4}$$

Notice that the row vector $\boldsymbol{c} = (\alpha t_1 + q c_1, \ldots, \alpha t_d + q c_d, \alpha/2^{\ell+1})$ belongs to L, since it can be obtained by multiplying the last row vector by α and then subtracting appropriate multiples of the first d row vectors. Since the last coordinate of this vector discloses the hidden number α, we call $\boldsymbol{c}$ the *hidden vector*. The hidden vector is very close to the (publicly known) row vector $\boldsymbol{v} = (v_1, \ldots, v_d, 0)$. By trying to find the closest vector to $\boldsymbol{v}$ in the lattice L, one can thus hope to find the hidden vector $\boldsymbol{c}$ and therefore the private key α. The article [14] presents provable attacks of this kind, and explains how the attack can be extended to bits at other positions. Such attacks apply to DSA but also to any El Gamal-type signature scheme (see for instance [15] for the case of ECDSA).

In our case, we simply build the previously mentioned lattice and the target vector $\boldsymbol{v}$, and we try to solve the closest vector problem with respect to $\boldsymbol{v}$, using the so-called embedding technique that heuristically reduces the lattice closest vector problem to the shortest vector problem (see [14] for more details). From each close vector candidate, we derive a candidate y for α from its last coordinate, and we check that the public key satisfies $\beta = g^y \pmod p$.

4 Results

As already mentioned in Section 3.3, using a glitch attack, we were able to generate 38 DSA signatures such that the least significant byte of the corresponding k was expected to be zero. Next, we applied the lattice attack of Section 3.4, using NTL's [18] implementation of Schnorr–Euchner's BKZ algorithm [17] with block size 20 as our lattice basis reduction algorithm. Out of the 38 signatures, we picked 30 at random to launch the lattice attack, and those turned out to be enough to disclose the DSA private key α after a few seconds on an Apple PowerBook G4. We only took 30 because we guessed from past experiments that 30 should be well sufficient.

To estimate more precisely the efficiency of the lattice attack, we computed success rates, by running the attack 100 times with different parameters. Results can be seen in Table 1. Because the number of signatures is small, the lattice dimension is relatively small, which makes the running time of the lattice attack negligible: for instance, on an Apple PowerBook G4, the lattice attack takes about 1 second for 25 signatures, and 20 seconds for 38 signatures. Table 1

Table 1. Experimental Attack Success Rates: n is the Number of Bytes Reset in k, and d is the Number of Signatures.

	Number d of Signatures															
$n \downarrow$	2	3	4	5	6	7	8	10	11	12	22	23	24	25	26	27
1											0%	10%	39%	63%	87%	100%
2								0%	69%	100%						
3					0%	69%	100%									
4				0%	100%											
5		0%	2%	100%												
6		0%	100%													
7	0%	96%	100%													
10	6%	100%														
11	100%															

shows how many signatures are required in practice to make the lattice attack work, depending on the number of least significant bytes reset in k. Naturally, there will be a tradeoff between the fault injection and the lattice reduction: when generating signatures with nonces with more reset bytes, the lattice phase of the attack will require less signatures. When only one signature is available, the lattice attack cannot work because there is not enough information in the single congruence used. However, if ever that signature is such that k has a large proportion of zero bytes, it might be possible to compute k by exhaustive search (using the congruence $\leftarrow \big(g^k \pmod p\big) \pmod q)$, and then recover α. From Table 1, we see that when two signatures are available, the lattice attack starts working when 11 bytes are reset in each k. When only one byte is reset in k, the lattice attack starts working (with non-negligible probability) with only 23 signatures.

It should be stressed that the lattice attack does not tolerate mistakes. For instance, 27 signatures with a single byte reset in k are enough to make the attack successful. But the attack will not work if for one of those 27 signatures, k has no reset bytes. It is therefore important that the signatures input to the lattice attack satisfy the assumption about the number of reset bytes. Hence, if ever one is able to obtain many signatures such that the corresponding k is expected (but not necessarily all the time) to have a certain number of reset bytes, then one should not input all the signatures to the lattice attack. Instead, one should pick at random a certain number of signatures from the whole set of available signatures, and launch the lattice attack on this smaller number of signatures: Table 1 can be used to select the minimal number of signatures that will make the lattice attack successful. This leads to a combination of exhaustive search and lattice reduction.

5 Countermeasures

The heart of this attack lies with the ability to induce faults that reset some of k's bits. Hence, any strategy allowing to avoid or detect such anomalies will help thwart the attacks described in this paper. Note that checking the validity of the signature after generation will not help, contrary to the case of fault attacks on RSA signatures [4]: the faulty DSA signatures used here are valid signatures which will pass the verification process. We recommend to use *simultaneously* the following tricks that cost very little in terms of code-size and speed:

- *Checksums* can be implemented in software. This is often complementary to hardware checksums, as software CRCs can be applied to buffers of data (sometimes fragmented over various physical addresses) rather than machine words.
- *Execution Randomization*: If the order in which operations in an algorithm are executed is randomized it becomes difficult to predict what the machine is doing at any given cycle. For most fault attacks this countermeasure will only slow down a determined adversary, as eventually a fault will hit the desired instruction. This will however thwart attacks that require faults in specific places or in a specific order.

  For instance, to copy 256 bytes from buffer a to buffer b, copy

$$b[f(i)] \leftarrow a[f(i)] \quad \text{for} \quad i = 0, \ldots, 255$$

where $f(i) = (x \times (i \oplus w) + y \pmod{256}) \oplus z$ and $\{x, y, z, w\}$ are four random bytes (x odd) unknown to the attacker.

– *Ratification counters and baits*: baits are small (< 10 byte) code fragments that perform an operation and test its result. A typical bait writes, reads and compares data, performs xors, additions, multiplications and other operations whose results can be easily checked. When a bait detects an error it increments an NVM counter and when this counter exceeds a tolerance limit (usually three) the card ceased to function.
– *Repeated refreshments*: refresh k by generating several nonces and exclusive-or them with each other, separating each nonce generation from the previous by a random delay. This forces the attacker to inject multiple faults at randomly shifting time windows in order to reset specific bits of k.

Finally, it may also be possible to have a real time testing of the random numbers being generated by the smart card, such as that proposed in the FIPS140-2. However, even if this is practical it may be of limited use as our attack requires very few signatures to be successful. Consequently, our attack may well be complete before it gets detected.

What is very important is that no information on k is leaked, and that k is cryptographically random.

6 Conclusion

We described a method for attacking a DSA smart card vulnerable to fault attacks. Similar attacks can be mounted on any other El Gamal-type signature scheme, such as ECDSA and Schnorr's signature. The attack consisted of two stages. The first stage dealt with fault injection. The second involved forming a lattice for the data gathered in the previous stage and solving a closest vector problem to reveal the secret key.

The attack was realised in the space of a couple of weeks and was made easier by the inclusion of peaks on the I/O. This information could have been derived by using power or electromagnetic analysis to locate the target area, but would have taken significantly longer. The only power analysis done during this attack was to note when the crypto-coprocessor started to calculate a modular exponentiation.

References

1. F. Bao, R. Deng, Y Han, A. Jeng, A. Narasimhalu and T. Hgair, Breaking Public Key Cryptosystems and Tamper Resistant Devices in the Presence of Transient Faults, 5-th Security Protocols Workshop, Springer-Verlag, LNCS 1361, pp. 115–124, 1997.

2. H. Bar-El, H. Choukri, D. Naccache, M. Tunstall and C. Whelan, *The Sorcerers Apprentice Guide to Fault Attacks*, Workshop on Fault Diagnosis and Tolerence in Cryptography in association with DSN 2004 - The International Conference on Dependable Systems and Networks, pp. 330–342, 2004.
3. E. Biham and A. Shamir, *Differential Fault Analysis of Secret Key Cryptosystems*, Advances in Cryptology - CRYPTO'97, Springer-Verlag, LNCS 1294, pp. 513–525, 1997.
4. D. Boneh, R. DeMillo and R. Lipton, *On the Importance of Checking Cryptographic Protocols for Faults*, Journal of Cryptology, Springer-Verlag, nol. 14, no. 2, pp. 101–119, 2001.
5. D. Boneh and R. Venkatesan, *Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes*, Advances in Cryptology - CRYPTO'96, Springer-Verlag, LNCS 1109, pp. 126–142, 1996.
6. E. Dottax, *Fault Attacks on NESSIE Signature and Identification Schemes*, NESSIE Technical Report, October 2002.
7. C. Giraud and E. Knudsen, *Fault Attacks on Signature Schemes*, Workshop on Fault Diagnosis and Tolerence in Cryptography in association with DSN 2004 - The International Conference on Dependable Systems and Networks, 2004.
8. J. Hoch and A. Shamir, *Fault Analysis of Stream Ciphers*, Cryptographic Hardware and Embedded Systems - CHES 2004, Springer-Verlag, LNCS 3156, pp. 240–253, 2004.
9. N. A. Howgrave-Graham and N. P. Smart, *Lattice Attacks on Digital Signature Schemes*, Design, Codes and Cryptography, vol. 23, pp. 283–290, 2001.
10. N. Joshi, K. Wu and R. Karri, *Concurrent Error Detection Schemes for involution Ciphers*, Cryptographic Hardware and Embedded Systems - CHES 2004, Springer-Verlag, LNCS 3156, pp. 400-412, 2004.
11. T. May and M. Woods, *A New Physical Mechanism for Soft Errors in Dynamic Memories*, Proceedings of the 16-th International Reliability Physics Symposium, April, 1978.
12. National Institute of Standards and Technology, FIPS PUB 186-2: Digital Signature Standard, 2000.
13. P. Q. Nguyễn, *Can we trust Cryptographic Software? Cryptographic Flaws in GNU Privacy Guard v1.2.3*, Advances in Cryptology - EUROCRYPT 2004, Springer-Verlag, LNCS 3027, pp. 555–570, 2004.
14. P. Q. Nguyễn and I. E. Shparlinski, *The Insecurity of the Digital Signature Algorithm with Partially Known Nonces*, Journal of Cryptology, vol. 15, no. 3, pp. 151–176, Springer, 2002.
15. P. Q. Nguyễn and I. E. Shparlinski, *The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces*, Design, Codes and Cryptography, vol. 30, pp. 201 – 217, 2003.
16. P. Q. Nguyễn and J. Stern, *The two faces of lattices in cryptology*, Cryptography and Lattices – CALC'01), Springer-Verlag, LNCS 2146, pp. 146–180, 2001.
17. C. P. Schnorr and M. Euchner, *Lattice basis reduction: improved practical algorithms and solving subset sum problems*, Math. Programming, vol. 66, pp. 181–199, 1994.
18. V. Shoup, *Number Theory C++ Library (NTL)*, `http://www.shoup.net/ntl/`