Deterministic DSA fault attack

Fabio Gritti
fabio1.gritti@mail.polimi.it

Sebastiano Mariani
sebastiano.mariani@mail.polimi.it

Professors: Gerardo Pelosi , Alessandro Barenghi

Contents

1 Introduction

In this paper we want to introduce two active side channel attacks against the deterministic version of the Digital Signature Algorithm (from now on DDSA) as specified in the RFC 6979[1].
These attacks can lead directly to a leak of the private key and therefore breaking the authenticity of the signatures created using this algorithm.

We will proceed in this way: first we provide some useful background in order to understand better the paper, then we explain the attacks in details and finally we present the feasibility of the attack by showing the time needed to break the DSA with some of the keysize currently available.

2 Background

2.1 The Digital Signature Algorithm

2.1.1 General overview

The DSA is one of three digital signature schemes specified in FIPS 186[2].
A digital signature scheme is an authentication mechanism that enables the creator of a message to attach a code that acts as a signature providing authenticity and non-repudation.
It usually consists of three algorithms:

- a *key generation algorithm* that outputs the private key and the corresponding public key.
- a *signing algorithm* that given a message and a private key produce the signature.
- a *signature verifying algorithm* that given a signature and a public key either accepts or rejects the message's claim to authenticity.

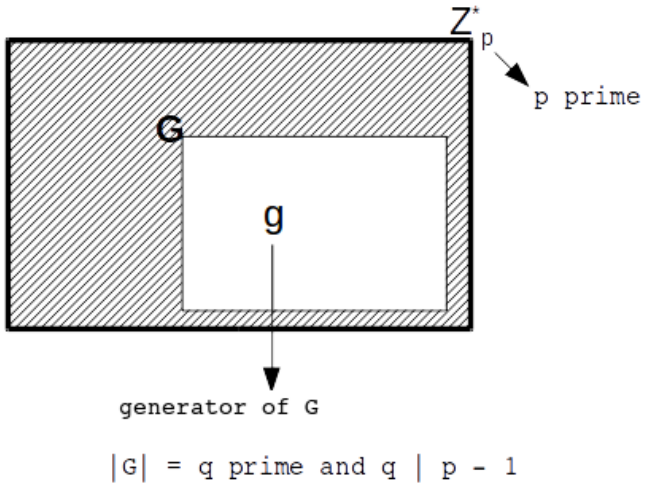
In order to be a sound digital signature schemes the following properties must hold:

- the authenticity of a signature generated from a fixed message and fixed private key can be verified by using the corresponding public key.

- it should be computationally infeasible to generate a valid signature for a party without knowing that party's private key.
- The private key MUST remain private, otherwise the authenticity of the signature created with that key is broken.
- The public key owner MUST be verifiable, otherwise we can't assure that we had receive a message from a particoular source.

2.1.2 DSA mathematical structure

The reference group category is $(\mathbb{Z}_p^*, \cdot)$, p prime s.t. $q \mid (p\text{-}1)$ with q also prime. The employed algebraic structure is then the multiplicative cyclic subgroup $G = <g>$ with publicly known generator g and order q.



FIPS 186-3 specifies the (L = $log_2(p)$ N = $log_2(q)$) length pairs of (1024,160),(2048,224),(2048,256) and (3072,256).

2.1.3 Cryptoscheme

Public Key: $k_{pub} = (p, q, g, g^s)$
Private Key: $k_{priv} = s \in \mathbb{Z}_q^*$

Signature Transformation:

$H(m) \in \{2, ..., q-1\}$
$k \overset{random}{\leftarrow} \mathbb{Z}_q^*$
r $\leftarrow ((g^l) \text{mod p}) \text{mod q}$
if r $= 0$ repeat with another random l
$s \leftarrow \text{k}^{-1} \cdot (H(m) - s * r)\ mod q$
if $s = 0$, repeat with another random k

$Signature =< s, r >$
$Send(m, Signature)$

Validation Check:

Receive <m,Signature>
if r,s $\notin \{1, ..., q-1\}$ reject the signature
Compute H(m)

Accept signature if:
$\text{u}_1 \leftarrow H(m) \cdot s^{-1} \text{mod q}$
$u_2 \leftarrow r \cdot s^{-1} \text{mod q}$
$(g^{u_1}((g^s)^{u_2}) \text{mod p mod q} == r?$

2.2 The Deterministic Digital Signature Algorithm

2.2.1 General overview

The step in the signing algorithm that require to generate a random parameter needs a great randomness in order to be secure; but digging with embedded system not always we have on board a sound RNG!
This is the crucial point for which many manufacturers decide to employ RSA cryptosystem rather than DSA/ECDSA to sign objects.

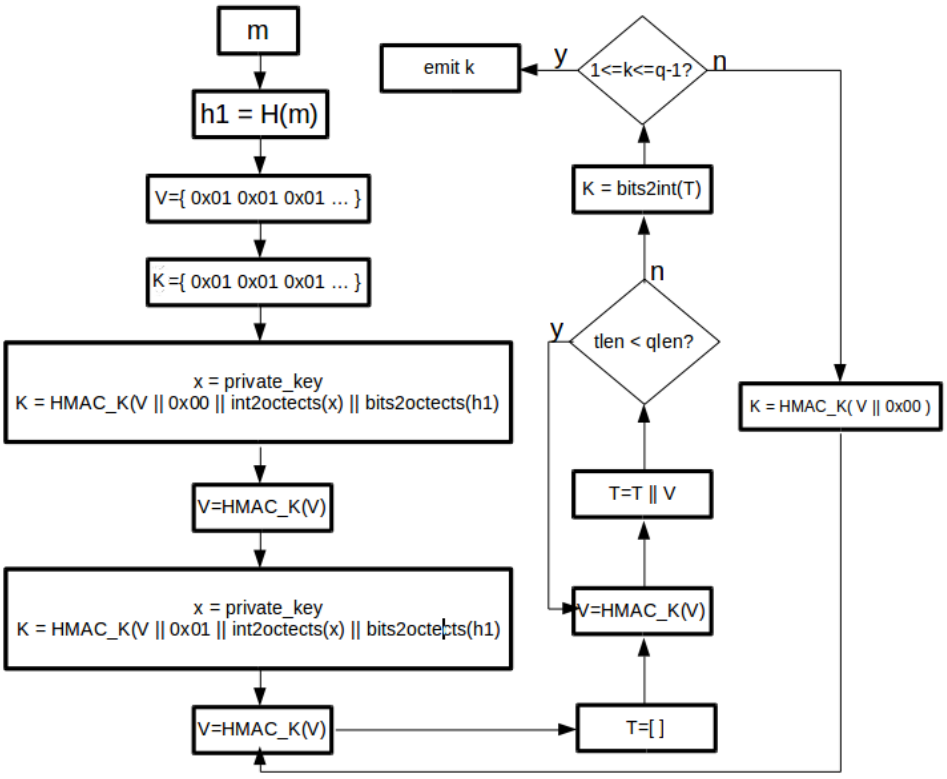
The Deterministic DSA proposed in the RFC 6979 try to make that step deterministic (and impossible to deduct without the private key) in order to increase the attractiveness of such system in the embedded world (smart card, credit card , console, ...).

Remember that we ,in any way, need a good RNG in order to generate

a strong private key. This proposal make deterministic the signing, NOT the generation of keys that MUST be non deterministic and made a priopri.

2.2.2 K deterministic generation

The generation of the deterministic k depends only from the message to sign and the private key.
According to the RFC , given the input message m, the following process is applied for the generation:



- qlen = binary length of the prime number q
- tlen = binary length of the array T
- The internal HMAC use the same hash function used in the h1=H(m)

2.3 Other digital signature algorithm

Chosing a different group and a different set of operations it is possible to build some variant of the DSA algorithm.The two principal cÃňvariants are:

- **dECDSA:** Is a variant of DSA which uses elliptic curve cryptography. With this type of cryptography is possible to achive the same security level, compared to the normal DSA algorithm, with a smaller key length.
- **EdDSA Bernstein:** The Edwards-curve digital sign algorithm is a digital signature scheme using a variant of Schnorr signature based on Twisted Edwards curves. It is designed for high performance while avoiding security problems that have surfaced in other digital signature schemes such as some side channel attack.

3 Cracking the Deterministic DSA

3.1 Differential fault analysis

The differential fault analysis is an active side channel attack that is performed in three steps:

- Obtain the correct signature of a message.
- Inject a fault during a second signing of the message and obtain the *faulty signature*.
- Correlate the correct signature with the faulty one and extract the private key.

In order to inject a fault we can use different techniques that range from using laser beam during the signing process, to voltage spike, insane overclocking etc..
Usually critical embedded systems like credit card implements different defenses against this fault injection techniques; this raise a lot the cost of an attack performed by an attacker.
Bypass this kind of defense is out of scope of this paper.

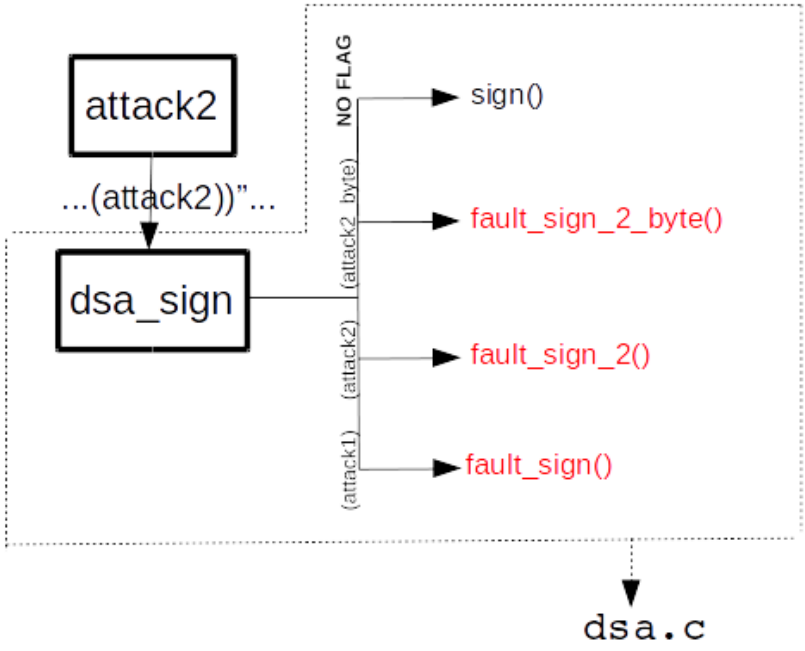
3.2 The attack in practice

We test our attack against the implementation of the DSA RFC 6979 in the GNU/libgcrypt[3] library. In order to inject the attack using the library we

added a flag during the building of the s-expression relative to the plaintext e.g.:

```
err = gcry_sexp_build(&ptx2, NULL, "(data (flags rfc6979) (hash
    %s %b) (attack2))" , "sha1", hash_len , digest);
```

When the function dsa-sign find that flag it redirects the flow to our faulty function that injects the desidered fault based on the type of the attack we want to perform.



The next two paragraphs describe the two attacks type and then we present the result obtained.

3.2.1 Damage the exponentation

Remembering from the DDSA scheme that we have to exponentiate the generator g at the power of k, we inject a bit fault into k during the exponentiation, obtaining $g^{\tilde{k}} = \tilde{r} = r \cdot g^{\pm 2^i}$.

```

[ FROM /libgcrypt/cipher/dsa.c fault_sign function ]

[...]

```

```
6  gcry_mpi_t one = mpi_set_ui(NULL, 1);
7
8  gcry_mpi_t e = mpi_new(qbits);
9
10 //calculate 2^bit_fault
11 mpi_mul_2exp(e, one, (unsigned long)bit_fault);
12
13 gcry_mpi_t k_tilde = mpi_new(qbits);
14
15 //damage the k
16 if( sign == 0 )
17    mpi_subm(k_tilde,k,e,skey->q);
18 else
19    mpi_addm(k_tilde,k,e,skey->q);
20
21 mpi_mul(sig_k, k_tilde, one);
22
23 //***************** FAULT *********************//
24 mpi_powm( r, skey->g, k_tilde, skey->p );
25 mpi_fdiv_r( r, r, skey->q );
26 //***************** FAULT *********************//
27
28
29 [ ... ]
```

The attacks proceeds as follows:

- Obtain the correct signature $s = k^{-1}(m + xr)$
- Obtain the faulty signature $\tilde{s} = k^{-1}(m + xr)$
- Write the system:

  $$\begin{cases} s = k^{-1}(m + xr) \\ \tilde{s} = k^{-1}(m + x\tilde{r}) \end{cases}$$

  with k and x are unknown.
- Solving,we obtain:

  $$\begin{cases} x \equiv_q \frac{(s-\tilde{s})m}{(r\tilde{s}-\tilde{r}s)} \\ k \equiv_q s^{-1}(mx + r) \end{cases}$$

This attack is very interesting because it doesn't need any bruteforce in order to discover the key: all the parameters that we need in the previous formula are available from the output of the correct and faulty algorithm.

The correctness of the attack is checked by creating a new dsa-key-pair based on the private key calculated and by signing a new plaintext with that key and then try to verify the signature with the original public key.

3.2.2 Damage the signature composition

For this attack we consider two level: a bit level and a byte level.

The bit level assume that with an active side channel attack we can surgically flip a single bit of k during the signature composition; this assumption is really optimistic and this precision can lead to a very fast cracking during the bruteforcing part of the algorithm.

The byte level is a more realistic assumption that we inject a fault of a byte inside the k; in this case the bruteforcing part of the attack will be slower , but the cost in order to inject the fault pratically is reduced.

3.2.2.1 Bit level fault

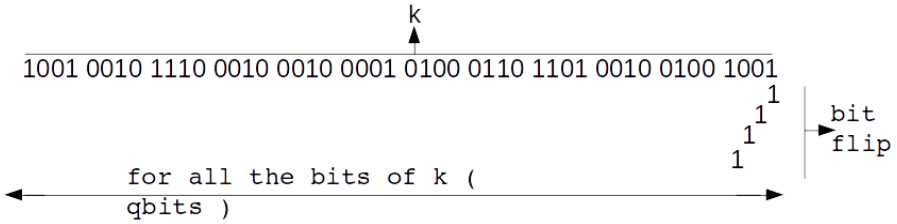
```
1
2  [ FROM /libgcrypt/cipher/dsa.c fault_sign_2 function ]
3
4  [...]
5
6  mpi_powm( r, skey->g, k, skey->p );
7  mpi_fdiv_r( r, r, skey->q );
8
9  gcry_mpi_t one = mpi_set_ui(NULL, 1);
10
11 //calculat 2^i ()
12 gcry_mpi_t e = mpi_new(qbits);
13 mpi_mul_2exp(e, one, (unsigned long) bit_fault);
14
15 gcry_mpi_t k_tilde = mpi_new(mpi_get_nlimbs(k));
16
17
18 //damage the k
19 if(sign==0)
20    mpi_subm(k_tilde,k,e,skey->q);
21 else
22    mpi_addm(k_tilde,k,e,skey->q);
23
24 //****************** FAULT *********************//
25
26 mpi_mul(sig_k, k_tilde, one);
```

```
27
28  kinv = mpi_alloc( mpi_get_nlimbs(k) );
29  mpi_invm(kinv, k_tilde, skey->q );
30
31  //***************** FAULT ********************//
32
33 [...]
```

The attacks with a bit fault proceeds as follows:

- Obtain the correct signature $s = k^{-1}(m + xr)$
- Obtain the faulty signature $\tilde{s} = \tilde{k}^{-1}(m + xr)$
- Notice that
  $\frac{s}{\tilde{s}} = \frac{k^{-1}(m+xr)}{\tilde{k}^{-1}(m+xr)}$
- In case of a negative fault $k - 2^i$ we have
  $\frac{s}{\tilde{s}} = \frac{k^{-1}(m+xr)}{(k-2^i)^{-1}(m+xr)}$
- In case of a positive fault $k + 2^i$ we have
  $\frac{s}{\tilde{s}} = \frac{k^{-1}(m+xr)}{(k+2^i)^{-1}(m+xr)}$
- calulating k for both the case we have:
  $k_{faultpos} = \frac{\tilde{s}2^i}{s-\tilde{s}}$
  $k_{faultneg} = \frac{\tilde{s}2^i}{\tilde{s}-s}$

- Notice that in this case we need a bruteforce in order to discover the right i (which bit has been flipped, 0<=i<=qbits) and so to finally calculate k



- After discovered the k, we can calculate the private key x as:
  $x = \frac{sk-m}{r}$

3.2.2.2 Byte level fault

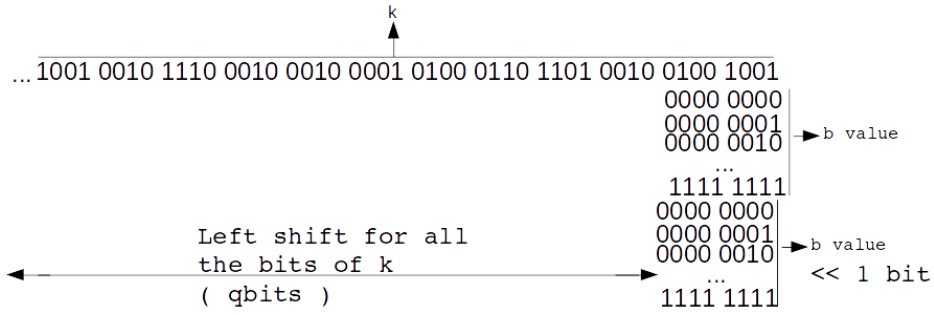
```
[ FROM /libgcrypt/cipher/dsa.c   fault_sign_2_byte function ]

[ ... ]

mpi_powm( r, skey->g, k, skey->p );
mpi_fdiv_r( r, r, skey->q );

gcry_mpi_t one = mpi_set_ui(NULL, value_fault);


gcry_mpi_t e = mpi_new(qbits);
mpi_mul_2exp(e, one, byte_fault);

gcry_mpi_t k_tilde = mpi_new(mpi_get_nlimbs(k));

if(sign==0)
   mpi_subm(k_tilde,k,e,skey->q);
else
   mpi_addm(k_tilde,k,e,skey->q);

//****************** FAULT *********************//

mpi_mul(sig_k, k_tilde, one);

kinv = mpi_alloc( mpi_get_nlimbs(k) );
mpi_invm(kinv, k_tilde, skey->q );

//****************** FAULT *********************//

[ ... ]
```

The attacks with a byte fault proceeds as follows:

- Obtain the correct signature $s = k^{-1}(m + xr)$
- Obtain the faulty signature $\tilde{s} = \tilde{k}^{-1}(m + xr)$
- Notice that
  $\frac{s}{\tilde{s}} = \frac{k^{-1}(m+xr)}{\tilde{k}^{-1}(m+xr)}$
- In case of a negative fault $k - b2^i$ we have
  $\frac{s}{\tilde{s}} = \frac{k^{-1}(m+xr)}{(k-b2^i)^{-1}(m+xr)}$
- In case of a positive fault $k + b2^i$ we have
  $\frac{s}{\tilde{s}} = \frac{k^{-1}(m+xr)}{(k+b2^i)^{-1}(m+xr)}$

- calulating k for both the case we have:
  $k_{faultpos} = \frac{\tilde{s}b2^i}{s-\tilde{s}}$
  $k_{faultneg} = \frac{\tilde{s}b2^i}{\tilde{s}-s}$

- Notice that in this case we need a bruteforce in order to discover the right b and i (which byte has been flipped, 0<=i<=qbits and the value of the fault b, 0<=b<=255) and so to finally calculate k. The index i is the value of the shift of the fault value b (we are trying all the possible value, for all the possible shift).



- After discovered the k, we can calculate the private key x as:
  $x = \frac{sk-m}{r}$

4 Results

We have tested these attack on 4 key length:

- p = 1024 bit q = 160 bit
- p = 2048 bit q = 224 bit
- p = 2048 bit q = 256 bit
- p = 3072 bit q = 256 bit

And with two scenarios for each length:

- optimal case: the fault happens on the first bit/byte of k
- worst case: the fault happens on the last bit/byte of k

4.1 Damage the exponentation

- Bit level Byte level:

  In this attack the distincton between bit level and byte level has no meaning because the time complexity is alway O(c) regardless the type of the fault injected. The private key is cracked in less than one second even if the fault is injected at the word level.

  These are the average time necessary in order to complete the attack

	Avg time
Length 1	2.241930 ms
Length 2	7.508261 ms
Length 3	8.846499 ms
Length 4	16.900461 ms

4.2 Damage the signature composition

- Bit level:

	Optimal case	Worst case
Length 1	4.953000 ms	295.578003 ms
Length 2	14.788000 ms	1547.551025 ms
Length 3	15.799001 ms	2021.688965 ms
Length 4	28.413000 ms	3562.698975 ms

- Byte level:

	Optimal case	Worst case
Length 1	132.716995 ms	75677.210938 ms
Length 2	479.232025 ms	290135.656250 ms
Length 3	555.675049 ms	447803.687500 ms
Length 4	972.052002 ms	841949.000000 ms

5 Conclusion

References

[1] https://tools.ietf.org/html/rfc6979

[2] http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

[3] https://www.gnu.org/software/libgcrypt/