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 present 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 this paper, then we explain the attacks in details and finally we present the feasibility of the attacks by showing the time needed to break the dDSA 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 given message and a given private key can be verified by using the corresponding public key.
- it should be computationally unfeasible to generate a valid signature for an user without knowing that users's private key.
- The private key MUST remains private, otherwise the authenticity of the signatures created with that key are 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.

In the FIPS 186-3[2] standardization the number of bits of the two prime numbers (L = $log_2(p)$ N = $log_2(q)$) are setted to: 1024|160, 2048|224, 2048|256 and 3072|256.

2.1.3 Cryptoscheme

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

In the following algorithm H stands for an hash function chosen at the be-

ginning of the transformation.

Algorithm 1: DSA signature transformation algorithm

Data: A message m to sign and a dsa key pair
Result: The signature of the given message m

```
r = 0;
s = 0;
h = H(m);
while r == 0 or s == 0 do
    k = rand() ∈ Z_q^* ;
    r = ((g^k) mod p ) mod q ;
    s = k^{-1} · (H(m) − x · r) mod q;
end
signature = <r,s>;
send(m,signature);
```

Algorithm 2: DSA verify algorithm

Data: A DSA signature $<r,s>$ to verify , a message m to be verified against the received signature and the dsa public key associated to the private one used to sign the message m
Result: Accept or not the signature

```
if r,s ∉ {1, ...., q − 1} then
    print "Signature rejected";
    return -1;
else
    h = H(m);
    u_1 = h·s^{-1} mod q ;
    u_2 = r·s^{-1} mod q ;
    if (g^{u_1}((g^s)^{u_2}) mod p mod q == r then
        print "Signature accepted";
        return 1;
    else
        print "Signature rejected";
        return -1;
    end
end
```

2.2 The Deterministic Digital Signature Algorithm

2.2.1 General overview

The step in the DSA signing algorithm that requires to generate a random parameter (a "true" random number, not a pseudorandom one) needs a great source of randomness in order to be secure; on normal PCs we can generate sound random numbers by exploiting unpredictable data like entropy, the exact time an user press keys on keyboard, the position of the mouse at a specific time, or any other kind of event.

Dealing with embedded systems is a problem from this point of view,in fact we don't have, on board, a real randomness source as before and so not a great RNG to generate unpredictable random numbers for our algorithm!
This is the crucial point for which many manufacturers decide to employ RSA cryptosystem, that doesn't require random number during the encryption transformation, rather than DSA/ECDSA, the latter in fact require by design a true random number for the encryption transformation.

The Deterministic DSA proposed in the RFC 6979[1] tries to "safely derandomize" the step in the algorithm that needs a true random number for the k parameter in order to increase the attractiveness of such system in the embedded world (smart card, credit card, console or any other kind of board). The security relies on whether the deterministic generation of k is indistinguishable from the output of a random oracle.

Remember that,in any way, we 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 and a private key, the following process is applied for the generation:

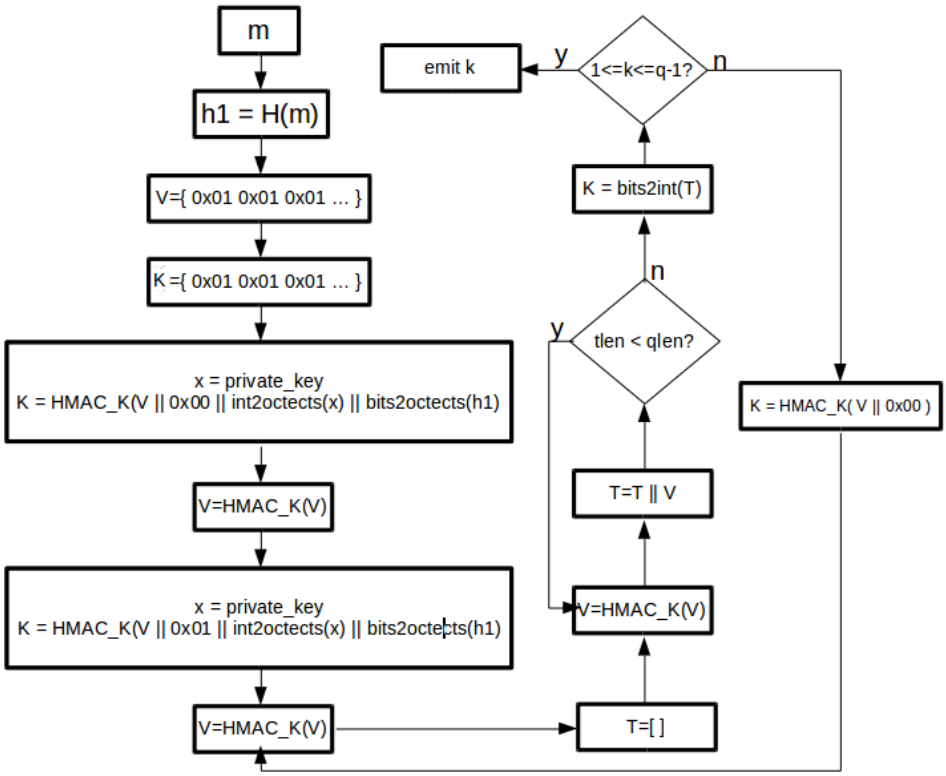

Figure 1: Generation of deterministic K parameter as specified in the RFC 6979

- 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)

This process has been selected in order to maximize as possible the dependency of the deterministic k from the message to sign and the private key used. The final goal is to make the deterministic generation of k indistinguishable from the output of a random oracle. The math structure behind this is out of scope of this paper, you can read more details in the RFC[1].

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 or insane overclocking.
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, our assumption is that the system is vulnerable to this kind of attacks.

3.2 The attacks

We test our attacks against the implementation of the RFC 6979[1] in the GNU/libgcrypt library[3]. Before start we are going to make a fast briefing about how you can use this library to create digital signature using dDSA. This step is fundamental in order to understand how we simulate the active side-channel attacks via software.
The GNU/libgcrypt library[3] makes heavily use of the so called S-expression[4]: a convenient way to parse and store data. For example if we want to generate a dsa key pair we have to write these few lines of code:

```
1
2 /* gcry_sexp_t is the data type used to store a S-expression */
3 gcry_sexp_t dsa_key_pair , dsa_parms;
4
5 /*
6 This function is used in order to build the S-expression
7 and store it in dsa_params. These parameters will
8 tell to the gcry_pk_genkey the options used to generate the key.
9 */
```

```
gcry_sexp_build(&dsa_parms,NULL,"(genkey(dsa (nbits 4:1024)))");

/*
Finally this is the function that will generate an
S-expression, stored in dsa_key_pair, that represent
the dsa key pair.
*/
gcry_pk_genkey(&dsa_key_pair,dsa_parms);
```

Now that we have a dsa key pair stored in the variable `dsa_key_pair` we can access its field simply:

```

/* get the generator of the group employed */
gcry_sexp_t g_param = gcry_sexp_find_token(dsa_key_pair,"g",0);

/* get the prime number 'p' employed */
gcry_sexp_t p_param = gcry_sexp_find_token(dsa_key_pair,"p",0);

/* get the prime number 'q' employed */
gcry_sexp_t q_param = gcry_sexp_find_token(dsa_key_pair,"q",0);

/* get the private key 'x' */
gcry_sexp_t x_param = gcry_sexp_find_token(dsa_key_pair,"x",0);

/* get the g^x parameter */
gcry_sexp_t y_param = gcry_sexp_find_token(dsa_key_pair,"y",0);
```

And signing a message is easy too:

```

/*
let's declare an mpi ( multiple precision number ) in order to
store the big numerical representation of the message to sign.
*/
gcry_mpi_t msg_digest;

/*
S-expression used in order to declare the plaintext to sign
and receive the ciphertext from the DSA signing algorithm.
*/
gcry_sexp_t ciphertext , plaintext;


/*
This is the SHA-1 of the message we want to sign.
in this case: "Hello world."
*/
const unsigned char* digest = (const unsigned char * )
     "e44f3364019d18a151cab7072b5a40bb5b3e274f";
```

```
20
21 /*
22 Specific function employed in the library that convert the
23 string representation of the message's digest in a multiple
24 precision number stored in the  variable 'msg_digest'.
25 */
26 gcry_mpi_scan(&msg_digest, GCRYMPI_FMT_USG, digest,
27                       strlen((const char*) digest), NULL);
28
29 /*
30 Creation of the S-expression relative to the data to sign.
31 Here you have to specify the flags rfc6979 in order to
32 make use of the deterministic DSA.
33 You can see this S-expression as the list of options and
34 parameters for the DSA signing transformation.
35 */
36 gcry_sexp_build(&plaintext, NULL, "(data (flags rfc6979) (hash %
    s %b))" , "sha1", 20 , msg_digest);
37
38 /*
39 Start the DSA signing transformation with the options
40 and parameters passed in the previous step and get
41 the result in the 'ciphertext' variable.
42 */
43 gcry_pk_sign(&ciphertext, plaintext, dsa_key_pair);
44
45 /*
46 If we want we can also verify here the signature
47 employing the publickey stored in teh variable
48 'dsa_key_pair'.
49 */
50 gcry_pk_verify (ciphertext, plaintext, dsa_key_pair);
```

We have presented how you can generate and use a pair of dsa key in order to sign a message using the dDSA specified in the RFC[1].

In order to simulate the attack using the library we added an "attack parameter" (in the next example *attack2*) during the building of the S-expression relative to the parameters of the DSA signing transformation:

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

We have then modified a bit the library (specifically the function *dsa-sign* in the file */libgcrypt/cipher/dsa.c*) in order to redirects the flow to our functions that injects the desidered fault based on the type of the attack we want to perform, we will call this a small "attack dispatcher". In total we added 3 functions:

- `fault_sign()`: when the flow is redirected to this function we trigger the attack 1 that will damage the exponentation (more details in the next section)
- `fault_sign_2()`: calling this function we trigger the attack 2 that will damage the signature composition (more details in the next section)
- `fault_sign_2_byte()`: this is a second version of the attack 2

If the "attack dispatcher" doesn't find any "attack parameter", the normal sign() function is called and we obtain the correct signature of the message.

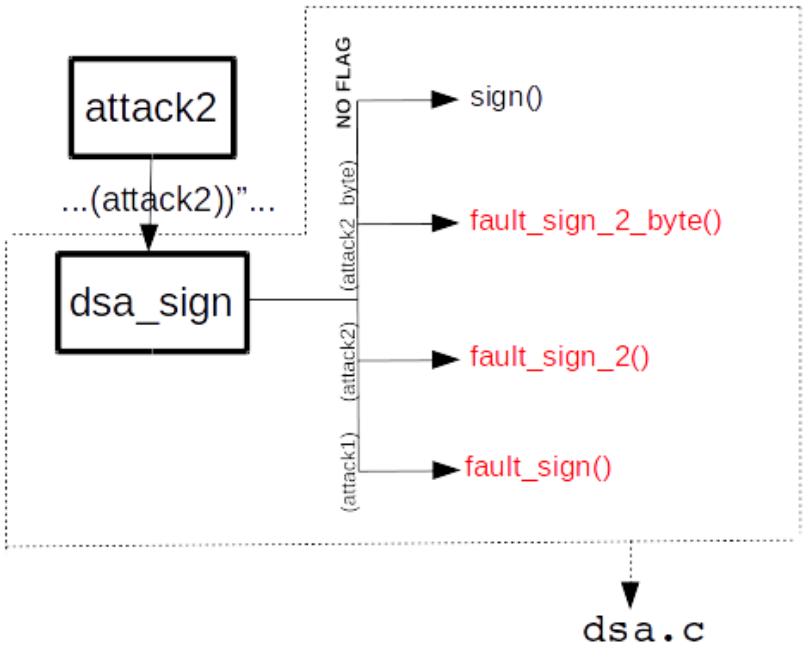

Figure 2: The schematization of the "attack dispatcher"

The next two paragraphs describe the two attacks, then we present the result obtained and finally some limits and considerations.

3.2.1 Damage the exponentation

Remembering from the line 6 of the DSA-signing algorithm presented in the section 2.1.3 that we have to exponentiate the generator g at the power of the deterministic generated k, we injected a bit fault into k during the exponentiation, obtaining: $g^{\tilde{k}} = \tilde{r} = r \cdot g^{\pm 2^i}$. Keep in mind that in the following formula with m we indicate the result of H(m).

The attack proceeds as follows:

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

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

  with k and x 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 fast and interesting because in order to calculate the private key x we need only the outputs from the two signatures (the parameters r and s from the correct one and the parameters r and s from the faulty one) and the message m passed in plaintext on the channel. The complexity of this attack is basically O(1) since it involves only the resolution of an equation.

We checked the attack's correctness 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 public key of the original dsa key pair.

3.2.2 Damage the signature composition

Remembering from the line 7 of the DSA-signing algorithm presented in the section 2.1.3 that we have to perform the multiplication s = $k^{-1} \cdot (H(m) - x \cdot r)$ mod q , we injected a fault into k during this multiplication (in this case

before the calculation of k^{-1} , but it would be the same flipping the final result k^{-1}), obtaining: $\tilde{s} = \tilde{k}^{-1}(m + xr)$.

For this attack we have to consider two type of possible injected fault: a bit level fault and a byte level one. The bit level assume that with an active side channel attack we can surgically flip a single bit of k during the signature composition, while the byte level is a more realistic assumption that we inject a fault of a whole byte inside the k. Other possible injected fault would be at WORD level,DWORD level, QWORD level, and the more generic fault with different bit flipped in different positions inside k (for example the MSB and the LSB in the most general case). As said for this attack we consider only the bit and byte injected fault model, we will generalize at the end for the others.In order to model a complete random and not guessable fault we consider both the cases of a positive bit/byte injection and negative bit/byte injection.

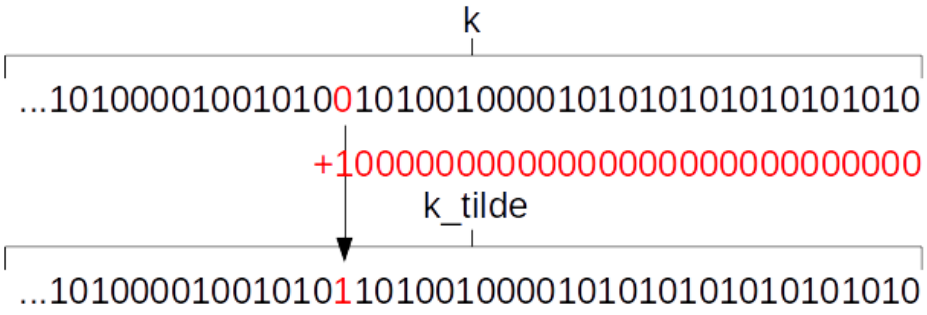

Figure 3: Attack2 with a positive injected bit fault

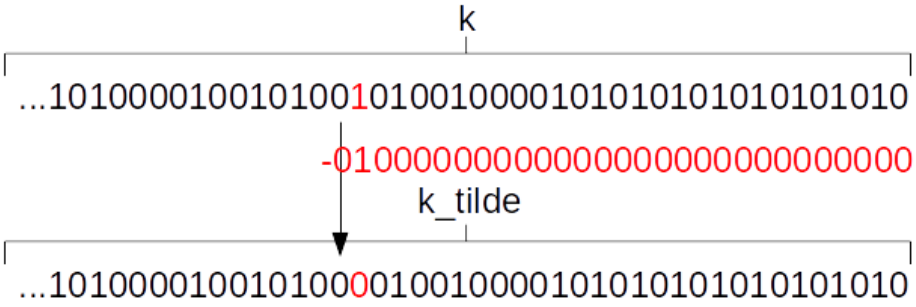

Figure 4: Attack2 with a negative injected bit fault

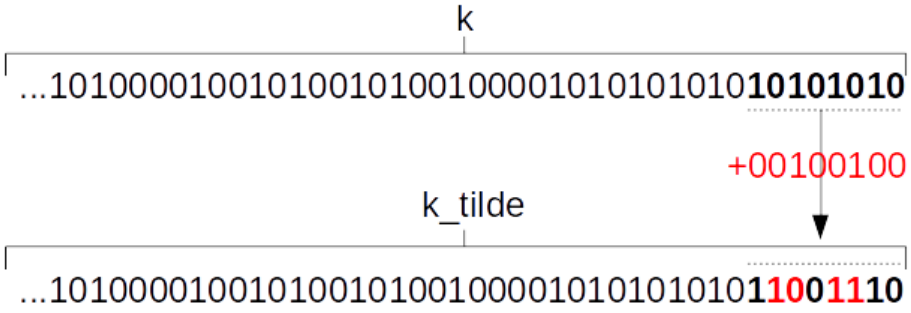

Figure 5: Attack2 with a positive injected byte fault

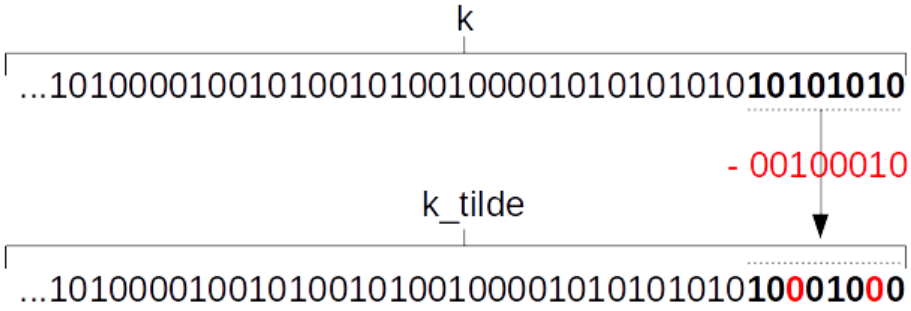

Figure 6: Attack2 with a negative injected byte fault

3.2.2.1 Bit level fault

The attacks with an injected 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 positive 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 negative 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:

  $k_{faultpos} = \frac{\tilde{s}2^i}{s-\tilde{s}}$
  $k_{faultneg} = \frac{\tilde{s}2^i}{\tilde{s}-s}$

- After having calculated k, we can derive the private key x as:

  $x = \frac{sk-m}{r}$

Notice that in this case we need the value of k in order to calculate x.Obviously we don't have it a priori, but we can discover it trying to bruteforce the exponent i of 2 in the formula $k_{faultpos} = \frac{\tilde{s}2^i}{s-\tilde{s}}$,so basically we calculate k for every possible i such that 0<=i<=qbits and we try to derive x employing the just calculated k with the formula $x = \frac{sk-m}{r}$, finally if our correctness check it's succesfull we can conclude that was the correct k and so this is the correct private key x calculated, otherwise we try with another i and

restart.

Algorithm 3: Attack 2, bit level

Data: A message m, the correct signature <r,s> , the faulty signature <r,$\tilde{s}$>

Result: The private key x of the dsa key pair employed to sign the message m

```
1  i=0;
2  //trying all the possibilities for positive faults;
3  while i <= qbits do
4  |  k_faultpos = s̃2^i / (s − s̃);
5  |  x = (sk − m)/r;
6  |  if correctness-check(x)==True then
7  |  |  print "dDSA private key cracked!";
8  |  |  return 1;
9  |  else
10 |  |  i++;
11 |  |  continue;
12 |  end
13 end
14 i=0;
15 //trying all the possibilities for negative faults;
16 while i <= qbits do
17 |  k_faultneg = s̃2^i / (s̃ − s) ;
18 |  x = (sk − m)/r;
19 |  if correctness-check(x)==True then
20 |  |  print "dDSA private key cracked!";
21 |  |  return 1;
22 |  else
23 |  |  i++;
24 |  |  continue;
25 |  end
26 end
27 print "Something went wrong";
28 return -1
```

The complexity of the algorithm is O(2n) where n=qbits.

3.2.2.2 Byte level 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:

  $k_{faultpos} = \frac{\tilde{s}b2^i}{s-\tilde{s}}$
  $k_{faultneg} = \frac{\tilde{s}b2^i}{\tilde{s}-s}$

- After discovered the k, we can calculate the private key x as:

  $x = \frac{sk-m}{r}$

  Notice that in this case, as before, 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).

Algorithm 4: Attack 2, byte level

Data: A message m, the correct signature <r,s> , the faulty signature <r,$\tilde{s}$>

Result: The private key x of the dsa key pair employed to sign the message m

```
1  i=0;
2  b=0;
3  //trying all the possibilities for positive faults;
4  while i <= qbits do
5  |  while b <= 255 do
6  |  |  k_faultpos = s̃b2^i / (s − s̃);
7  |  |  x = (sk − m) / r;
8  |  |  if correctness-check(x)==True then
9  |  |  |  print "dDSA private key cracked!";
10 |  |  |  return 1;
11 |  |  else
12 |  |  |  b++;
13 |  |  |  continue;
14 |  |  end
15 |  end
16 |  b=0;
17 |  i++;
18 end
19 i=0;
20 b=0;
21 //trying all the possibilities for negative faults;
22 while i <= qbits do
23 |  while b <= 255 do
24 |  |  k_faultneg = s̃b2^i / (s̃ − s);
25 |  |  x = (sk − m) / r;
26 |  |  if correctness-check(x)==True then
27 |  |  |  print "dDSA private key cracked!";
28 |  |  |  return 1;
29 |  |  else
30 |  |  |  b++;
31 |  |  |  continue;
32 |  |  end
33 |  end
34 |  b=0;
35 |  i++;
36 end
37 print "Something went wrong";
38 return -1
```

The complexity of the algorithm is O(2 ·255 ·n) where n=qbits.

3.3 Generalizing the fault model

During the explanation of the attack 2 we said that we considered only two injection models: the bit level and the byte level, but since often you can't control precisely where the injected fault will finish, a fault can be spread in a position larger than a byte, for example can be at WORD level, DWORD level, or at the QWORD level.

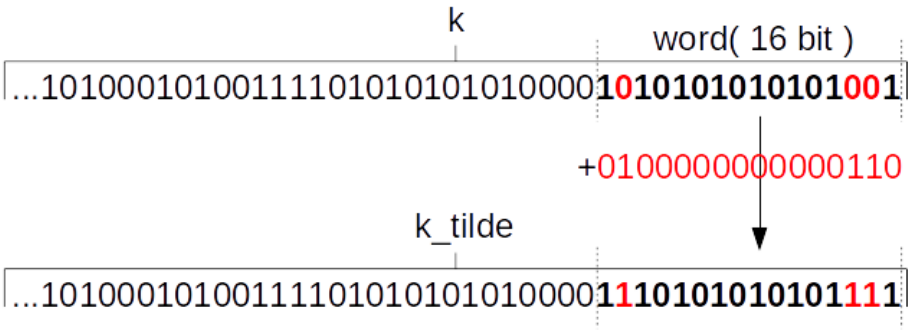

Figure 7: Attack2 with a positive injected word fault

The algorithm for the attack remains the same, but in this case the nested loop described in the algorithm-4 at lines 5 and 23, will be: *while* $b <= 2^{16}$ and since qbits depends on the length of the key we have a complexity of $O(2^{16} \cdot qbits)$. With the longest key size considered (p=3072,q=256) we will have a number of iteration of $2^{16} \cdot 2^8 = 2^{24}$; in order to compute this we would need a better parallelized algorithm using a framework like OpenCL[5].

The same thing applies to the DWORD level (32 bit) and the QWORD level (64 bit) the first will have a max. number of iteration of: $2^{32} \cdot 2^8 = 2^{40}$, the latter will have a max. number of iteration of: $2^{64} \cdot 2^8 = 2^{72}$.

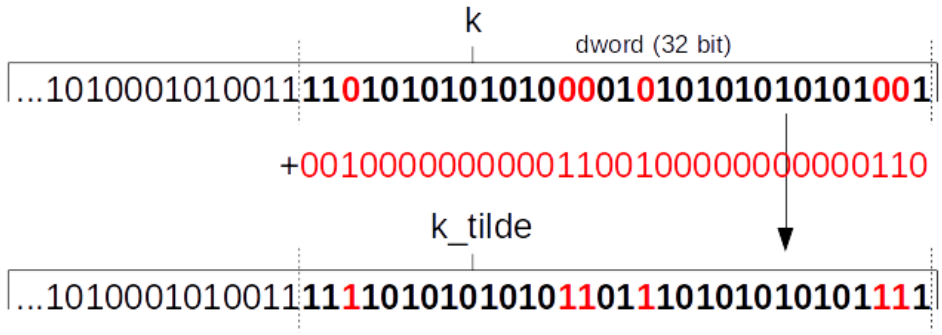

Figure 8: Attack2 with a positive injected dword fault

Notice that a fault injected at QWORD level is the max. that we can handle, in fact handling a wider fault will require inside our attack's algorithm a number of iteration greater that 2^{80} that is unfeasible; by the way a fault at the byte/WORD level is good and feasible.

Finally we can say that if we have a fault that had hitten two bits that can't be grouped in a byte/WORD/DWORD/QWORD our attack2 is not feasible, but as said before the injection of a fault at WORD level is usually feasible; If we can't inject a fault that we can handle we can in any case switch to attack1 since it doesn't need this step as explained previously.

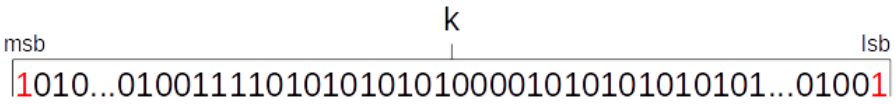

Figure 9: Attack2 will fail in these cases

4 Experimental results

We have tested these attacks using an Intel i5 and 2GB ram with Xubuntu 14.04. The implementation of the DDSA algorithm as specified in the RFC[1] is taken from the libgcrypt library version 1.6.4 [3].

In order to test the attacks you have to copy our file dsa.c inside the folder /libgcrypt/cipher/ and the recompile the whole library.
The experiments are performed against 4 key length (expressed in the form pbits|qbits): 1024|160 , 2048|224 , 2048|256 , 3072|256.

For the attack 2, during the fault injection, we have to consider a completely random fault that range from an optimal case to the worst possible case.(As explained before this model is not necessary for the attack 1 because its success is independent from where the fault happens)

- optimal case: the fault happens on the LS[Bit/Byte] of k; in this case we are lucky and we have the fastest possible cracking.

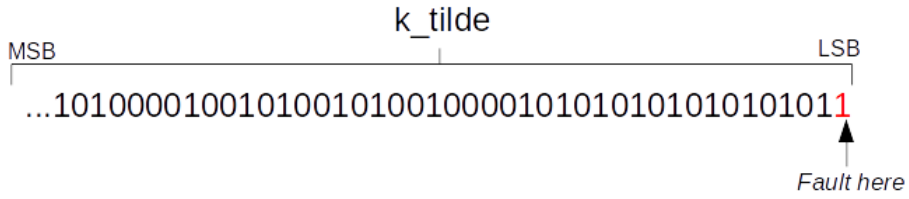

Figure 10: Attack2, bit level with the best scenario

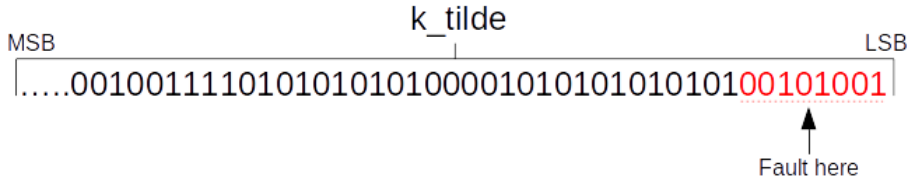

Figure 11: Attack2, byte level with the best scenario

- worst case: the fault happens on the MS[Bit/Byte] of k; in this case we are not lucky during the fault injection and we have the slowest possible cracking.

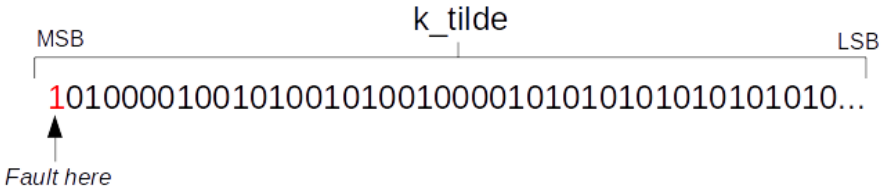

Figure 12: Attack2, bit level with the worst scenario

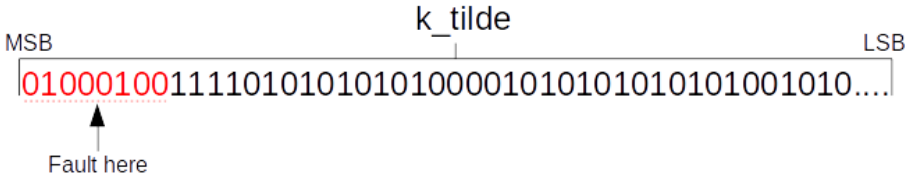

Figure 13: Attack2, byte level with the worst scenario

For this reason the attack's time estimated with a single run would be an imprecise measure.In order to give a better estimation of the attack's time we have randomized in the library code where the fault happen and taken the average time produced by 30 attack's rounds.The registered times include also the generation of the original signature and the validation step of the attack.

4.1 Damage the exponentation

In this attack the distinction between bit/byte/WORD/DWORD/QWORD level has no meaning because the time complexity is always O(1) regardless the type of the fault injected.
These are the average times necessary in order to complete the attacks,for each key length:

Table 1: Average times over 30 runs registered for attack 1

Key length	Time [ms]
1024/160	7
2048/224	11
2048/256	12
3072/256	23

4.2 Damage the signature composition

These are the time necessary in order to complete the attack at the bit level and byte level.

Table 2: Average times over 30 runs registered for attack 2 at the bit level

Key length	Time [ms]
1024/160	171
2048/224	572
2048/256	779
3072/256	2094 $\sim$ 2 sec.

Table 3: Average times over 30 runs registered for attack 2 at the byte level

Key length	Time [ms]
1024/160	44267 $\sim$ 44 sec.
2048/224	212596 $\sim$ 3.5 min.
2048/256	8398991 $\sim$ 2.20 hour.
3072/256	19778534 $\sim$ 5.4 hour.

5 Limits

As presented before, in order to work the attacks presented need the parameters r and s from the two signatures (the correct one and the faulty one) of the same message m; all this parameters are sent in plain over the untrusted

channel. The attacks fail if any of the previous condition aren't satisfied: we can't retreive the parameter r or s, we can't retreive the message m, or we can't sign the message m two times.

Given that, a possible mitigation of these attack would be, for example, to protect the signature sended over the channel, or the message sent, or why not both, with the public key of the destination; in this scenario we can't retreive enough parameters in order to satisfy the equations that calculate the private key.

6 Conclusion

According to the RFC[1], "the determinism of the algorithm described in this note may be useful to an attacker in some forms of side-channel attacks". With this paper we prove this fact by simulating two active side channel attacks via software that can retreive the private keys of a dsa key pair with any length. The descriptions of these attacks can be easily applied to a real scenario where the attacked system doesn't implement good countermeasures against side-channel attacks. Finally we proposed a possible easy mitigation against these attacks.

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/`

[4] `http://rosettacode.org/wiki/S-Expressions`

[5] `https://www.khronos.org/opencl/`

[6] `https://en.wikipedia.org/wiki/Digital_signature`