Active Side-Channel Attack against Deterministic DSA

Sebastiano Mariani, Lukas Dresel

May 27, 2018

Abstract

This paper will examine how to perform an active side channel attack against the deterministic version of the Digital Signature Algorithm as specified in the RFC 6979 [2]. This attack allows an attacker to directly leak the private key used in the signature. We provide a proof of concept exploit implementation in Python which demonstrates the effectiveness of the attack by leaking the private key with constant linear time complexity.

1 Introduction

In this paper we will examine fault side-channel attacks against the deterministic Digital Signature Algorithm implementation described in RFC 6979 [2].
We present an implementation of one such attack which can lead to leaking the private key and consequently breaking the authenticity guarantees of the signatures since they can then be forged.

This paper is structured as follows: In Section 2 we go over background information required to understand both deterministic DSA and our proposed attack. The attack itself will be explained in detail in Section 3 and we show its feasibility by implementing a proof-of-concept which can break dDSA with some of the common key sizes and analyze its running time.

2 Background

2.1 The Digital Signature Algorithm

The digital signature algorithm (DSA) is specified as one of three signature schemes in FIPS 186 [1]. Digital signature schemes are authentication mechanisms which can provide authenticity and non-repudiation of a message by appending a code to the message called the signature for the respective author. In general, there are 3 algorithms that compose a signature scheme:

1. The *key generation algorithm* which creates both private and public key components

2. The *signature algorithm* which takes both the message and the private key and outputs the signature for that message.

3. The *verification algorithm* which when given message and signature using the public key determines whether or not the message was signed using the appropriate private key, and thereby accepts or rejects it.

A general digital signature scheme should have the following properties:

1. Without the appropriate private key it should be infeasible to generate a correct signature for any message which would be accepted by a specific public key

2. Private keys must remain confidential, otherwise the authenticity guarantees of the scheme would be broken

3. the authenticity of the signature for a given message generated with a given private key can be verified with the corresponding public key.

4. The owner of a given public key must be verifiable, otherwise a message can't be guaranteed to have come from the correct source.

2.2 DSA

DSA is usually performed using the group $(\mathbb{Z}_p^*, \cdot)$, p and q prime s.t. q — (*p-1*). The employed algebraic structure is then the multiplicative cyclic subgroup G with publicly known generator g and order q.

The FIPS 186-3 standard specifies the number of bits of the prime numbers (L, N) = ($log_2(p)$, $log_2(q)$) as any of the following: (1024, 160), (2048, 224), (2048, 256) and (3072, 256).

2.2.1 Algorithms

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

The algorithms are defined using a chosen hash function H. This function must be a cryptographic hash function and should be assumed to be known to the attacker.

Algorithm 1: DSA signature algorithm

Data: Message m to sign and a DSA key pair as specified above
Result: Signature of message m

```
1 r = 0;
2 s = 0;
3 h = H(m);
4 while r == 0 or s == 0 do
5 |   k = rand() ∈ Z_q^* ;
6 |   r = ((g^k) mod p ) mod q ;
7 |   s = k^{-1} · (H(m) − x · r) mod q;
8 end
9 return (r, s);
```

Algorithm 2: DSA signature verification algorithm

Data: DSA signature (r,s) to verify, message m to be verified, DSA public key for the presumed author of message m
Result: Whether the signature was accepted or rejected

```
1 if r,s ∉ {1, ...., q − 1} then
2 |   return "Rejected"
3 else
4 |   h = H(m);
5 |   u_1 = h · s^{-1} mod q ;
6 |   u_2 = r · s^{-1} mod q ;
7 |   if (g^{u_1}((g^x)^{u_2}) mod p mod q == r then
8 |   |   return "Accepted"
9 |   else
10|   |   return "Rejected"
11|   end
12 end
```

2.3 The Deterministic Digital Signature Algorithm

2.3.1 Motivation for deterministics DSA

The random number used in step 5 of Algorithm 1 is required the be a cryptographically secure random number with high entropy for the algorithm to be secure. Since normal computers are fully deterministic and cannot generate true entropy they usually use user input like times of key strokes or the positions of the cursor over a certain time-period to collect the required entropy.

This dependency can not however be met on more recent devices like IoT devices or other embedded electronics because users are not available and therefore cannot be used to harvest entropy. For this reason most manufacturers of these devices rely on cryptographic systems without the need for high-degree random numbers at runtime. This explains the prevalence of the RSA cryptosystems in embedded devices over DSA or ECDSA as they require randomness during the construction of the signatures as

opposed to only during the generation of the key pairs.

To make DSA more attractive for embedded systems like smart devices, key cards, etc., RFC 6979 [2] proposes the deterministic usage of DSA (dDSA). It proposes to create the key k from the private key k_{priv} and the message m instead of creating it randomly, therefore derandomizing the signing step. The security of dDSA then relies on the indistinguishability of k from a random oracle.

While deterministic DSA eliminates the need for randomness in the signing step, the creation of the key pairs of course still requires randomness and so the keys on these kinds of systems must be created beforehand.

2.3.2 Deterministic generation of parameter k

The algorithm for generating the parameter k from the message and the private key is illustrated in Figure 1. The parameters are

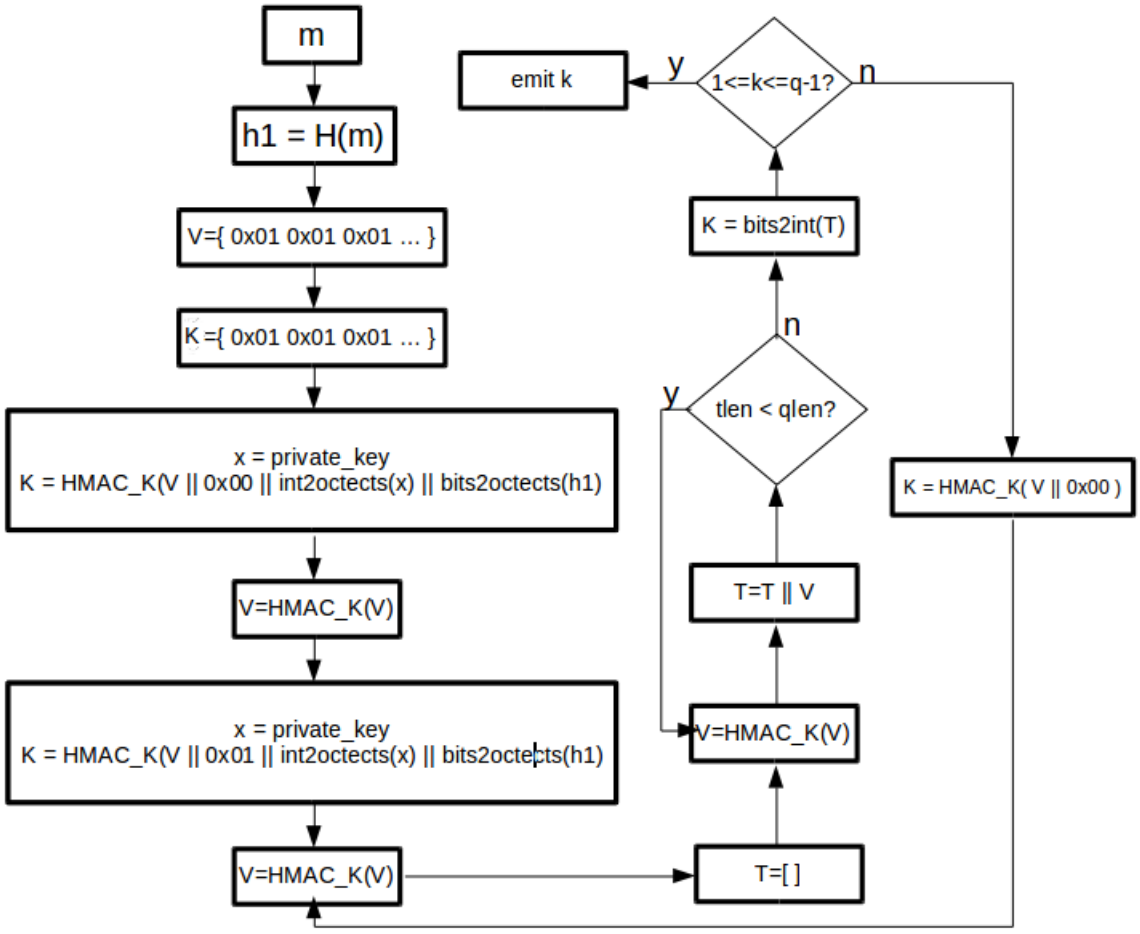

Figure 1: Generation of deterministic K parameter as specified in the RFC 6979

- *qlen* is the binary length of q
- *tlen* is the binary length of array T

- HMAC_K uses the same hash function as the other algorithms, H

3 Breaking Deterministic DSA

3.1 Differential fault analysis

Differential fault analysis is an active side channel attack. It is performed in three steps:

1. Obtain a correct signature s for message m
2. During a second signing of the message m a fault is injected to obtain a *faulty signature* $\hat{s}$.
3. Obtain the private key k_{priv} using s and $\hat{s}$.

An attacker would like to use attacks like this against critical embedded systems, e.g. credit cards, access cards, etc. in a variety of ways. These include, but are not limited to, exposing parts of the hardware to intense laser beams, spiking the voltage of the chips at the right time or overclocking the chips at the correct time.

While defenses against these techniques exist and significantly raise the cost of these types of attacks, they are not very heavily deployed depending on the field the hardware is used for.

We operate under the simplifying assumption that these kinds of defense are not deployed on the target system, as breaking them is out of the scope of this project.

3.2 The attack

3.2.1 Attacking the exponentiation

In the signature algorithm in Section 2.2.1 on line 6 the generator g is raised to the power of the deterministically generated k. By injecting a bit fault during this exponentiation we obtain instead $g^k = \tilde{r} = r \cdot g^{\pm 2^i}$.

The attack then consists of the following steps:

- Correct signature of H(m): $s = k^{-1}(H(m) + xr)$
- Faulty signature of H(m): $\tilde{s} = k^{-1}(H(m) + x\tilde{r})$
- Then
  1. $s \equiv_q k^{-1}(m + xr)$
  2. $\tilde{s} \equiv_q k^{-1}(m + x\tilde{r})$

  with k and x unknown.
- Solving for x and k gives

1. $x \equiv_q \frac{(s-\tilde{s})m}{(r\tilde{s}-\tilde{r}s)}$
2. $k \equiv_q s^{-1}(mx + r)$

By using this attack we can calculate the private key component x using only the outputs of two signing operations of a plaintext message m. Solving the above equations is possible in constant linear time complexity of O(1).

Using our implementation of both the algorithms and the attack in Python we verified our attack by creating a new key pair, signing a message once without injecting a fault and another time with the bitfault included. After recovering the private key k_{priv} we recreated a key pair for it and signed a new message m_{new} with it. We could then verified that the signature for m_{new} generated with the recovered key pair was accepted by the verification algorithm when given the old key pair.

References

[1] National Institute of Standards and Technology. Digital signature standard (dss). 2013.

[2] T. Pornin. Deterministic usage of the digital signature algorithm (dsa) and elliptic curve digital signature algorithm (ecdsa). 2013.