Project Report : Key Extraction Attack on ECDSA using Hidden Number Problem

Aman Jain Saiteja Nangunoori Sathvik Reddy Kollu Pavanbhargav Tangirala

April 20, 2019

Abstract

This work aims to explore how to extract the 256 bit private key used in ECDSA to sign messages by exploiting a previously unexplored part in the ECDSA signing algorithm based on a previous work. This attack leaves the newer implementations of various cryptographic libraries (such as OpenSSL 1.1.0h) vulnerable. In the rest of this report, we first explain the software vulnerability leaking the information via cache side channels followed by the implementation details of the attack. We finally analyse the results obtained and critique the work.

1 Introduction

The work presented here is based on the paper - "Return of the Hidden Number Problem : A Widespread and Novel Key Extraction Attack on ECDSA and DSA" by Keegan Ryan, NCC Group published in TCHES '19 (1).

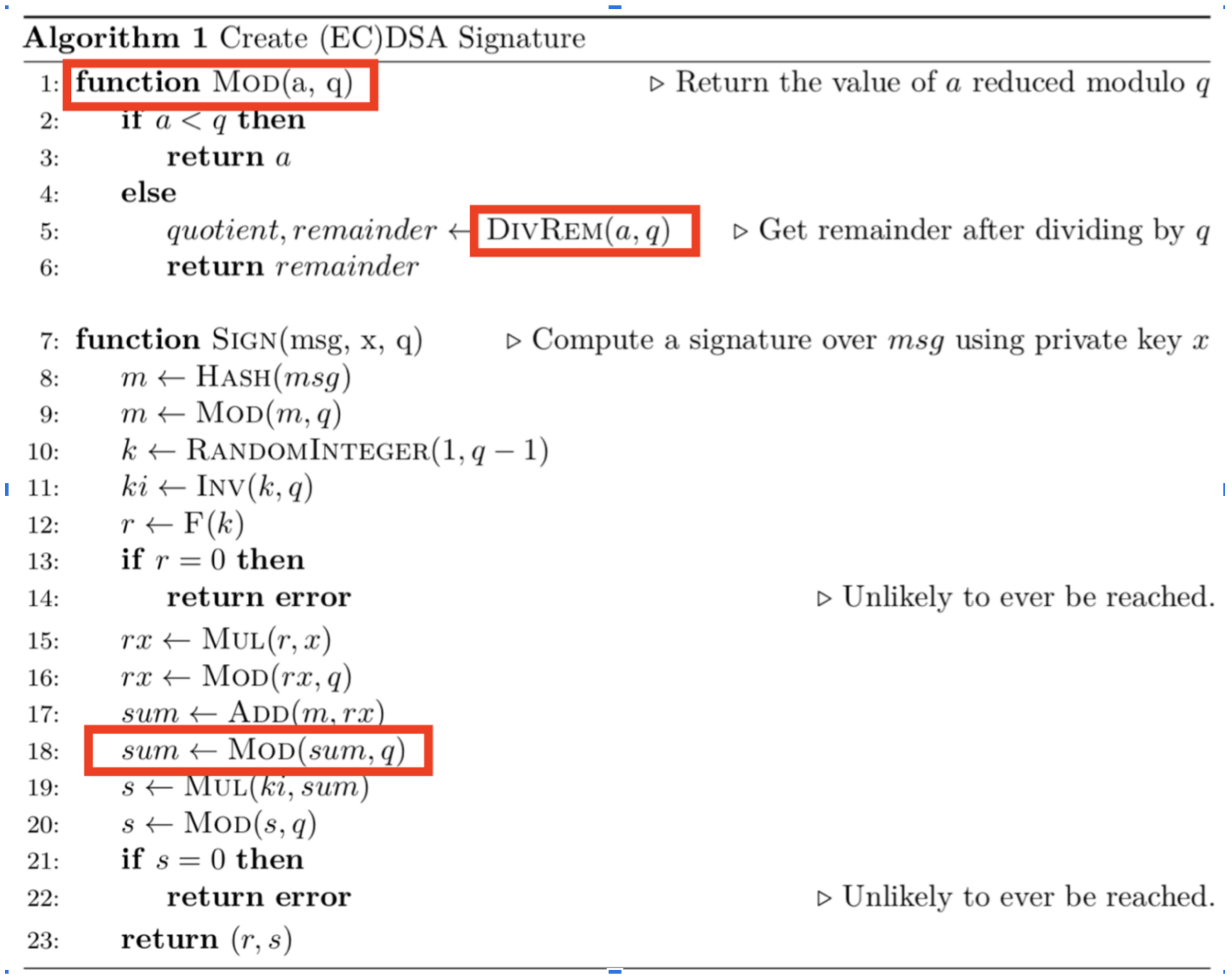
1.1 Overview

There have been various attacks in the past to extract the private key of ECDSA and DSA algorithms. Previous attacks such as (2) target the computation of modular exponentiation with a per-message ephemeral secret. This involves a sequence of modulo square and multiply operations which, if leaked by cache-based side channel attacks, enables an adversary to obtain the DSA private key.
Despite the mitigations implemented by several cryptographic libraries to defend against these attacks, the paper presents a novel side-channel attack against ECDSA. The attack targets a common implementation pattern that is found in many cryptographic libraries. As many as half of the libraries that were tested in the paper exhibited the vulnerable pattern. The software vulnerability targeted for the side channel attack is explained in the next section. The attack comprises of two parts - a

cache side channel attack to leak some information followed by solving a hidden number problem to completely extract the private key.

2 Software Vulnreability

The most common implementation of ECDSA signing algorithm is presented in 1. Only necessary helper functions are shown for the sake of clarity.



Algorithm 1 Create (EC)DSA Signature

```
 1: $\boxed{\textbf{function } \textsc{Mod}(\text{a, q})}$          ▷ Return the value of $a$ reduced modulo $q$
 2:     if $a < q$ then
 3:         return $a$
 4:     else
 5:         $quotient, remainder \leftarrow \boxed{\textsc{DivRem}(a, q)}$   ▷ Get remainder after dividing by $q$
 6:         return $remainder$

 7: function Sign(msg, x, q)          ▷ Compute a signature over $msg$ using private key $x$
 8:     $m \leftarrow \textsc{Hash}(msg)$
 9:     $m \leftarrow \textsc{Mod}(m, q)$
10:     $k \leftarrow \textsc{RandomInteger}(1, q - 1)$
11:     $ki \leftarrow \textsc{Inv}(k, q)$
12:     $r \leftarrow \text{F}(k)$
13:     if $r = 0$ then
14:         return error                                 ▷ Unlikely to ever be reached.
15:     $rx \leftarrow \textsc{Mul}(r, x)$
16:     $rx \leftarrow \textsc{Mod}(rx, q)$
17:     $sum \leftarrow \textsc{Add}(m, rx)$
18:     $\boxed{sum \leftarrow \textsc{Mod}(sum, q)}$
19:     $s \leftarrow \textsc{Mul}(ki, sum)$
20:     $s \leftarrow \textsc{Mod}(s, q)$
21:     if $s = 0$ then
22:         return error                                 ▷ Unlikely to ever be reached.
23:     return $(r, s)$
```

Figure 1: Pseudocode for ECDSA signing algorithm

It is important to note here that the MOD function does not run in constant time. If the argument (a) is less than q, it returns directly without calling DivRem function. Otherwise DivRem function is called to find the remainder. Cache based side channel attack can be used to leak this information whether DivRem function was called or not. Hence, at line 18, the range of sum (= m + rx mod q) can be leaked via side channel attack.

3 Hidden Number Problem

3.1 Theory

The HNP poses the problem of recovering an unknown scalar in a prime field when only partial information is known about multiples of the scalar. Formally in the HNP, one has knowledge of a number of multipliers $t_1, ..., t_d \in F_q^*$ for known prime q. For each sample i,one also knows MSB $_{l_i,q}(\lfloor t_i\alpha \rceil_q)$ for some fixed but unknown $\in [1, q-1]$. The goal is to recover hidden number . The HNP can identically be expressed as a system of inequalities:

$$|\lfloor t_i\alpha \rceil_q - u_i|_q < q/2^{l_i+1} \text{ for all i} \in 1, ..., d. \tag{1}$$

We can solve for α by forming a CVP (closest Vector Problem) with lattice as B and target vector u where $u = (2^{l_1+1}u_1, ..., 2^{l_d+1}u_d, 0)$. In B,u each dimension (column) is scaled proportionally to the inverse of the bounds(2^{l_i+1}) as CVP solvers weigh all dimensions equally.

$$B = \begin{bmatrix} 2^{l_1+1}q & & & 0 \\ & . & & . \\ & & . & . \\ & & 2^{l_d+1}q & 0 \\ 2^{l_1+1}t_1 & . & . \;\; 2^{l_d+1}t_d & 1 \end{bmatrix}$$

There exits some vector $x \in Z^{d+1}$ such that $xB = (2^{l_1+1}\lfloor u_1\alpha \rceil_q, ..., 2^{l_d+1}\lfloor u_d\alpha \rceil_q, \alpha)$, and this lattice vector is close to vector u as guaranteed by the HNP inequalities. We hope that solving the CVP with lattice $L(B)$ and vector u will yield lattice vector xB, since the last coordinate is the hidden value α.

But here we converted this instance of CVP to an instance of SVP by constructing a new lattice $B^{'}$ as following:

$$B^{'} = \begin{bmatrix} \mathbf{B} & \mathbf{0} \\ \mathbf{u} & 1 \end{bmatrix}$$

The above lattice basis is reduced by performing LLL followed by BKZ. The second shortest vector of reduced basis will be often of the form $(xB - u, -q)$ and the last coordinate of xB is α, the hidden number.

3.2 Converting the problem into HNP

As discussed above, in the ECDSA signing algorithm the range of sum (= m + rx mod q) can be leaked via side channel attack. This is a known plaintext attack, hence the value of m is known.

Also, r is known as it a part of the signature. Thus, knowing the range of sum (less than or greater than q) we need to find the value of x (the private key).Next, we constraint the inequations further to make them of the form a hidden number problem. Sign function does not return a signature unless r and s are non-zero.

Since $r \neq 0, \lfloor rx \rfloor_q \in [1, q-1],$ and since $s \neq 0, m + \lfloor rx \rfloor_q \notin \{0, q\}.$

If the side channel reveals that $\lfloor rx \rfloor_q \in [0, q-1]$, then we can conclude -

$$\begin{aligned}
& m + \lfloor rx \rfloor_q \in [0, q-1] \\
\Rightarrow \quad & \lfloor rx \rfloor_q \in [-m, q-m-1] \cap [1, q-1] = [1, q-m-1] \\
\Rightarrow \quad & \lfloor rx \rfloor_q - \frac{q-m}{2} \in [1 - \frac{q-m}{2}, \frac{q-m}{2} - 1] \\
\Rightarrow \quad & |\lfloor rx \rfloor_q - \frac{q-m}{2}|_q < \frac{q-m}{2}
\end{aligned}$$

Otherwise if it reveals that $\lfloor rx \rfloor_q \notin [0, q-1]$, then we can conclude -

$$\begin{aligned}
& m + \lfloor rx \rfloor_q \notin [0, q-1] \\
\Rightarrow \quad & m + \lfloor rx \rfloor_q \in [q+1, 2(q-1)] \\
\Rightarrow \quad & \lfloor rx \rfloor_q \in [q-m+1, 2q-2-m] \cap [0, q-1] = [q-m+1, q-1] \\
\Rightarrow \quad & |\lfloor rx \rfloor_q \in [q-m+1, q-1] \\
\Rightarrow \quad & \lfloor (q-r)x \rfloor_q \in [1, m-1] \\
\Rightarrow \quad & \lfloor (q-r)x \rfloor_q - \frac{m}{2} \in [1 - \frac{m}{2}, \frac{m}{2} - 1] \\
\Rightarrow \quad & |\lfloor (q-r)x \rfloor_q - \frac{m}{2}|_q < \frac{m}{2}
\end{aligned}$$

The above result (inequation based on the range of m) is converted to HNP inequation where t is r (or $(q-r)$), α is x, u is $\frac{q-m}{2}$ (or $\frac{m}{2}$), l_i is $\log(q/(q-m))$ (or $\log(q/m)$). By calculating HNP inequation for each signature, we will get set of HNP inequations using which the basis B, vector u are constructed. The hidden number α is calculated by reducing the basis $B^{'}$ (constructed as mentioned above using B,u) by applying LLL followed by BKZ reductions and finding last but one coordinate of second shortest vector of the reduced basis.

mitigation

4 Implementation Details

4.1 Attack Scenario

This work makes a few assumptions about the environment being attacked. In a vulnerable environment, client sends several messages to the server (in the clear) and the server (victim here) uses a private key to sign the messages. The attacker observes the resulting signatures and knows the messages being signed. Additionally, the attacker needs to run on the same machine as the server and use side channel information to deduces the private key used by the server to sign the messages. We emulated the scenario by creating server and attacker applications running on the same machine. The server runs on local-host and listens to a particular port. To act as a client, the attacker chooses random message values and sends to the sever to get signed. The rest is same as described above.

4.2 Side Channel Attack

To leak the information about the range of $m + \lfloor rx \rfloor_q$, a Flush + Reload attack is performed on the libcrypt.so.1.1 library of the OpenSSL 1.1.0g which contains the the ECDSA signing code of the OpenSSL library. This attack uses Mastik library (3) for the implementation of Flush + Reload attack. Following five offsets within the library are monitored :

1. "0x0f4e70" : Entry point of ECDSA_do_sign_ex — instruction 1
2. "0x0f4582" : Call to BN_mod_add_quick when computing $m + \lfloor rx \rfloor_q$ — instruction 2
3. "0x0a4b42" : Middle of BN_usub — instruction 3
4. "0x0a4b9f" : End of BN_usub, called from BN_mod_add_quick when the sum of arguments exceed the modulus — instruction 4
5. "0x0ed180" : End of ECDSA_SIG_free — instruction 5

The first offset is monitored in order to detect when the signature process began so the attacker could collect Flush+Reload samples until the last offset is reached. The second offset is used to detect when BN_mod_add_quick is called. Two offsets (third and fourth) within BN_usub are also monitored. If they are accessed by the victim, it implies that value of $m + \lfloor rx \rfloor_q$ is greater than q and vice versa. The pattern observed when $m + \lfloor rx \rfloor_q < q$ is shown below.

1. instructions 1,3,4 : not accessed
2. instructions 2,5 : accessed

If side channel information shows above pattern then in almost 100 percent cases the correct truth value of $m + \lfloor rx \rfloor_q$ is also less than q. But if above pattern is not observed we can't guarantee that $m + \lfloor rx \rfloor_q > q$. This means false positives are very low but there may be considerable amount of false negatives. So we only considered those samples for which above pattern is observed which ensures that the samples collected are almost 100 percent correct.

4.3 Lattice reduction and solving the HNP

After forming lattice from the information acquired from side channel attack as described in section 3.2 , the lattice is reduced using LLL reduction followed by BKZ reduction. The block size used while BKZ reduction is 20. The second smallest vector in the lattice i.e the second row of the reduced matrix is taken. The second last element in this row will be equal to our required hidden number.

4.3.1 Tradeoff between BKZ and LLL

Generally BKZ reduction is more accurate but slower when sample size is large. LLL is fast but it doesn't completely reduce the Lattice. Percentage accuracy is very low if we use only LLL. So instead we reduced using LLL first and then applied BKZ on this reduced matrix. But later we also tested by using only BKZ and no LLL. Even this gave good accuracy but almost took same time as LLL followed by BKZ. One probable reason for this could be that since the lattice size we are using(around 100) is not very large the differences in time taken to reduce using BKZ or LLL is not that significant.

5 Analysis

In this attack, the side channel leaks very less information, it only tells if the value of $m + \lfloor rx \rfloor_q$ is greater or less than q. Thus, the number of bits revealed by a signature is very less ($= \log(q/(q-m))$ or $\log(q/m)$). Hence, this attack requires thousands of signatures to extract the 256 bit private key. Compare this with the previous attacks (2) where side channel reveals a lot of bits and as a result require only a few signatures to extract the private key. But these attacks no longer work in the recent implementations of OpenSSL and other cryptographic libraries.
We performed experiments on solving the HNP and analysed the tradeoff between known bits and the number of signatures required to achieve a given success rate. The result obtained are shown in the figures below.

We can observe from these figures that to get almost 100% success rate, with known bits greater than or equal to 5, we need to collect only around 50 samples but it requires around 58,000 signatures. Hence, the lattice formed is small and takes very less amount of time to solve. With known

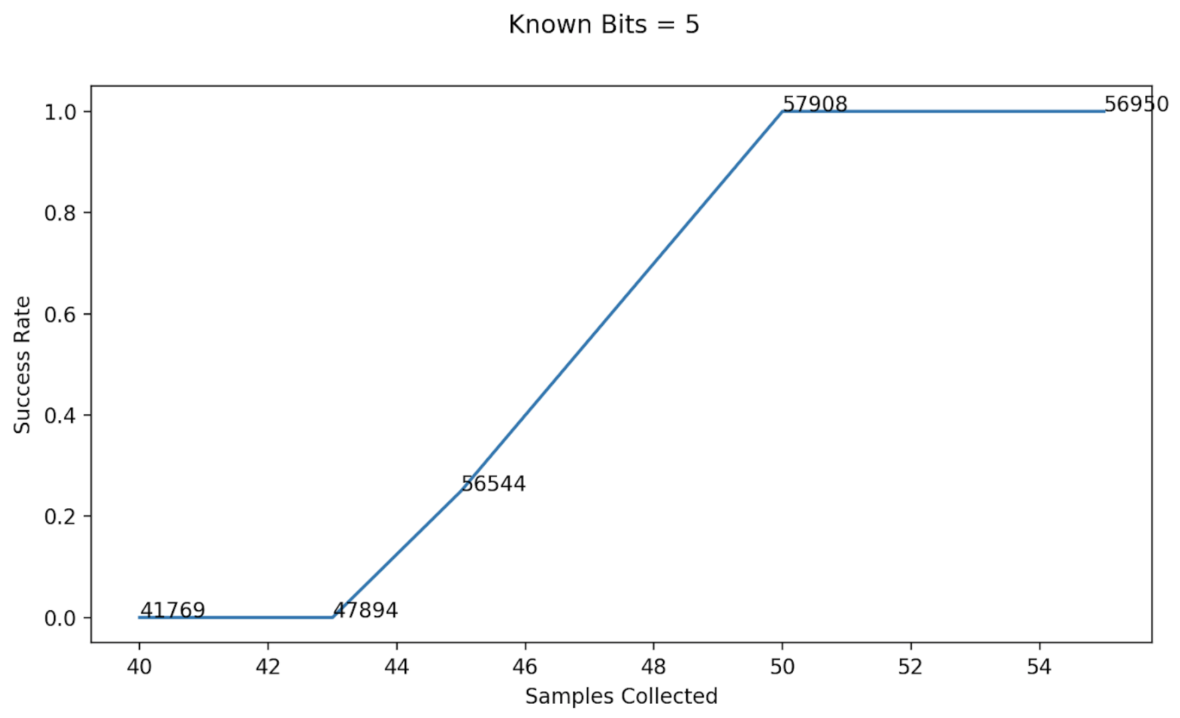


Figure 2: Variation of success rate with number of samples collected, for known bits >= 5 (labels denote the number of signatures required to collect these samples)

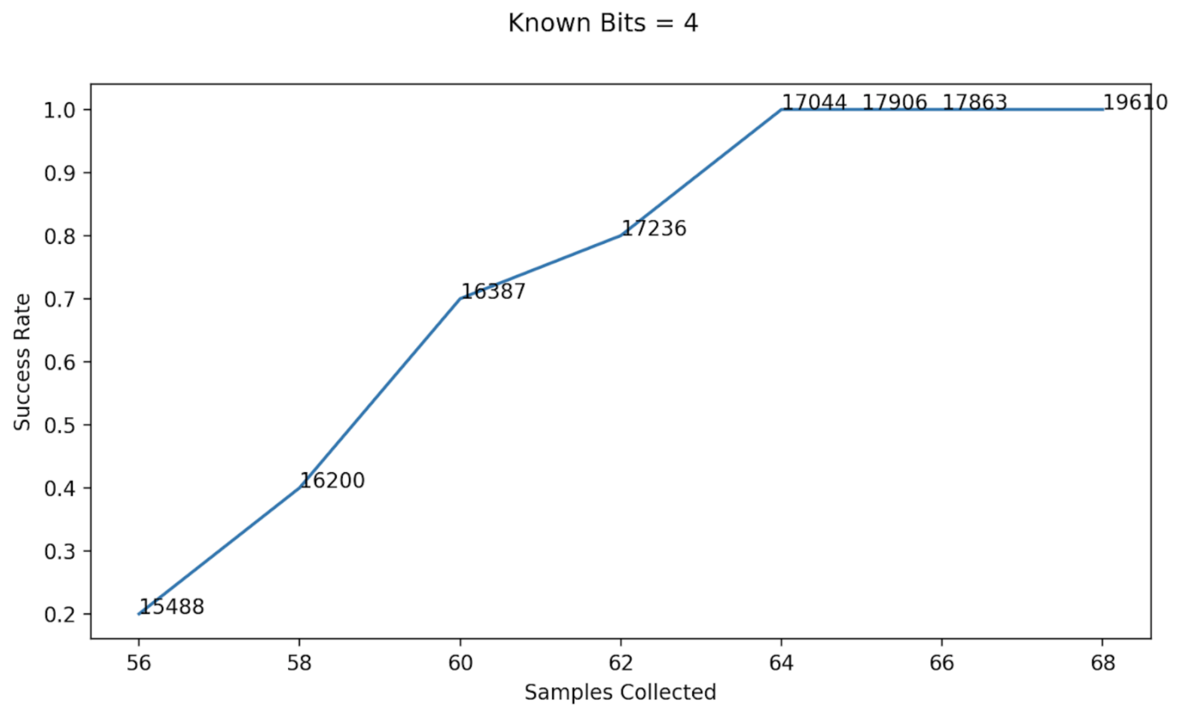


Figure 3: Variation of success rate with number of samples collected, for known bits >= 4 (labels denote the number of signatures required to collect these samples)

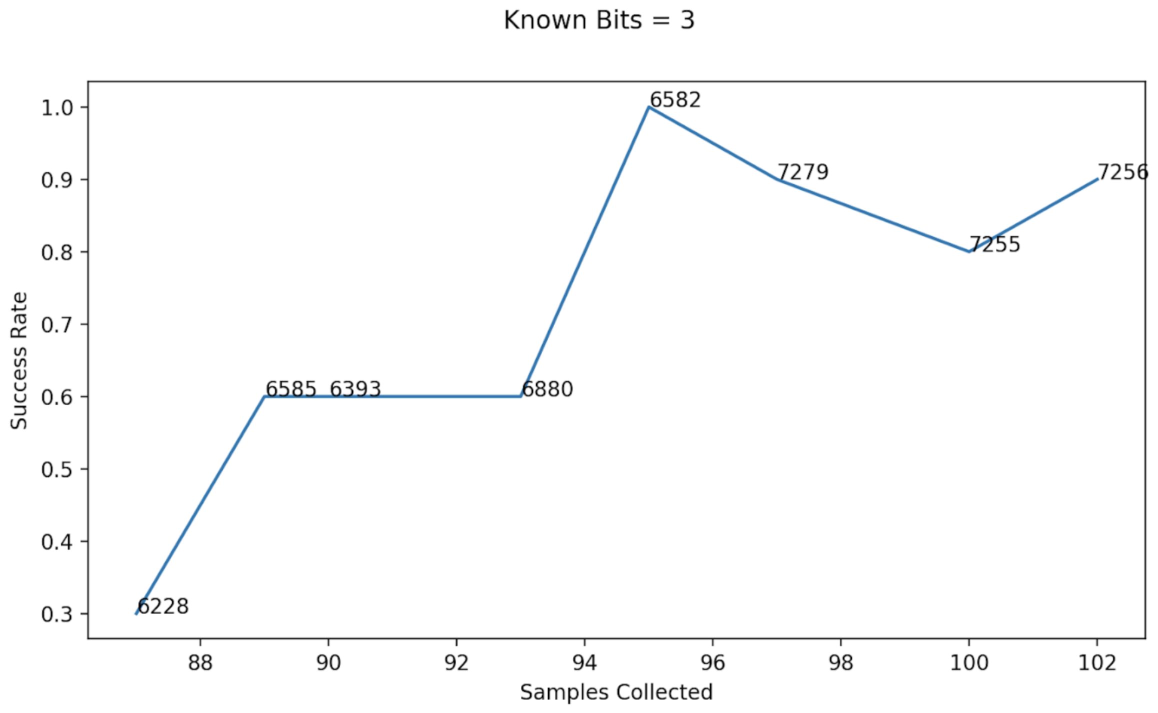


Figure 4: Variation of success rate with number of samples collected, for known bits >= 3 (labels denote the number of signatures required to collect these samples)

bits greater than or equal to 4, we need to collect around 65 samples but it requires around 17,000 signatures. But with known bits greater than or equal to 3, it is hard to assure 100% success rate even with large number of samples, since a small amount of error always persist. With even less number of known bits, the success rate is very low.

Another observation from the above figures is that the number of samples required to get almost 100% success rate is around $\frac{\text{number of bits in the key (=256 here)}}{\text{minimum number of known bits}}$. In other words, the total number of known bits from all the samples should be equal to the number of bits in the key.

6 Personal Section

6.1 Challenges in understanding the paper

The process of converting an instance of CVP to SVP was written without any clear explanation in the paper, which was quite unintutive at first. We had to refer to other papers to understand the conversion. The author didn't focus on the algorithms which he used such as LLL and BKZ, which are quite complex and requires a lot of background to get a deep understanding of the algorithms. Apart from these, the paper was easy to follow.

6.2 Challenges in our implementation and their addressal

We faced challenges in managing the python libraries with their dependencies & versions and were unable to install a python library, fpylll as it had dependencies issues with a system library. So, we addressed this by using Anaconda environment to manage python packages.
But this came with it's own pitfalls, the Anaconda environment did not support version 1.1.0g of OpenSSL package (the version vulnerable to this attack). So, we ran server outside the environment to use system's libcrypt.so file (which contains the the ECDSA signing code of the OpenSSL library). Even then, the server did not use this particular libcrypt.so file. To force the server to do so, we exported LD_PRELOAD path appropriately to preload that particular libcrypt.so file. All this has been explained in detail in code manual inside Appendix.

6.3 Interesting ideas in the paper

The attack described in the paper exposes a new surface for side channel attack in the implementation of ECDSA and DSA algorithms that was previously unexplored. This attack leaves the newer implementations of various cryptographic libraries (such as OpenSSL 1.1.0h) vulnerable. As many as half of the libraries that were tested in the paper exhibited the vulnerable pattern. The attacks in the past such as (2) target the computation of modular exponentiation with a per-message ephemeral secret for side channel attack, but this has been patched in the earlier versions of OpenSSL and other libraries. However, this paper targets the implementation of MOD function.

6.4 Critique of work

The paper provides a novel surface for side channel attack on the ECDSA signing algorithm. It leaves as many as half of the libraries vulnerable to the attack. Also, the author provides thorough results and explainations in the paper.
We feel that there was not much need for applying LLL reduction (which is done to reduce the computation time taken for solving HNP) as the time difference among these two methods (with and without LLL) is found out to be negligible in our situation, where the basis size is not large enough to create much difference.

6.5 Extensions and enhancements

The method we used to solve HNP is not error tolerant. It fails even if one in-equation out of given in-equations is wrong. A more error-tolerant approach to solving the HNP such as Bleichenbacher's solution can be used. But there are no good algorithms which solves the problem in less time.

7 Conclusion

In this report, we explored how to extract the 256 bit private key used in ECDSA to sign messages by exploiting a previously unexplored part in the ECDSA signing algorithm based on a previous work. We implemented the attack and compared the reults with that in the paper. We also performed experiments on solving the hidden number problem and analysed the tradeoff between known bits and the number of signatures required to achieve a given success rate. Finally, we also studied the mitigations of the attack.

References

[1] Keegan Ryan. *Return of the Hidden Number Problem : AWidespread and Novel Key Extraction Attack on ECDSA and DSA*. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2019.

[2] Jiji Angel, R Rahul, C Ashokkumar, Bernard Menezes. *DSA signing key recovery with noisy side channels and variable error rates*. International Conference on Cryptology in India, 2017.

[3] Yuval Yarom. *Mastik: A Micro-Architectural Side-Channel Toolkit*. Cryptographic Hardware and Embedded Systems, 2016. `https://cs.adelaide.edu.au/~yval/Mastik/`

Appendix

Code Manual

The code consists of two python scripts - attacker.py and server.py and a c file flushreload.c. The flushreload.c file performs flush+reload on the 5 addresses mentioned in section 4.2. The server.py just accepts messages on a port, signs them and sends back. The attacker.py first generates random messages, sends them to server to get them signed, executes flushreload.c to perform side channel, store valid samples to finally solve the HNP and extract the private key.

Installation

The following packages need to be installed for the code to work -

1. Mastik - download from `https://cs.adelaide.edu.au/~yval/Mastik/`
2. apt install pkg-config
3. apt install libfplll-dev
4. Following python 2.7 packages -
   (a) Cython
   (b) cysignals
   (c) fpylll
   (d) cryptography

   It might be convenient to install these packages using Anaconda environment as we faced a lot of trouble installing fpylll manually. More information on how to use anaconda to manage python packages can be found here - `https://www.digitalocean.com/community/tutorials/how-to-install-the-anaconda-python-distribution-on-ubuntu-16-04`

Running the code

First make following changes to the code for it to be able to work on your machine -

1. Change the shebang in attacker.py and server.py to point to anaconda's python and system's python respectively (If packages are not installed via anaconda, change shebang in both files to point to system's python)
2. Change line 51 in attacker.py to /path/to/libcrypt.so.1.1/provided/by/us

3. You may need to change the THRESHOLD global variable in flushreload.c as it varies across machines

Perform the following steps to run the attack -

1. Generate keypair and set their privilege level -
   (a) sudo rm -f private.pem public.pem
   (b) openssl ecparam -genkey -name prime256v1 -noout -out private.pem
   (c) openssl req -new -x509 -key private.pem -out public.pem
   (d) chmod 666 public.pem
   (e) chmod 400 private.pem
   (f) sudo chown root:root private.pem
2. Build Flush+Reload binary -
   gcc -Imastik/src -Lmastik/src -o flushreload flushreload.c -lmastik
3. In one terminal do the following -
   (a) export LD_PRELOAD=$LD_PRELOAD:/path/to/libcrypt.so.1.1/provided /by/us
   This ensures that the cryptographic libraries find this particular libcrypt file in path before others and so uses this
   (b) sudo ./server.py &
4. In another terminal do -
   ./attacker.py numsamples knownbits
   Here, numsamples and knownbits denote the number of samples required to be collected by the attacker and the minimum known bits derived from a signature to consider it to formulate the HNP respectively.