Cache based Side Channel Attack

Master Thesis Project Report

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology
by
Astha Jada & Achala Bhati
(153050027 & 153050056)

Supervisor:
Bernard L. Menezes

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076 (India)

25 June 2017

Approval sheet

This dissertation entitled Side Channel Attack by Astha Jada and Achala Bhati is approved for the degree of Master of Technology in Computer Science and Engineering from IIT Bombay.

Examiners

_____

_____

Supervisor

_____

Chairman

_____

Date: _____

Place: _____

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

_____

(Signature)

_____

(Name of the student)

_____

(Roll No.)

Date: _____

Abstract

This document explains cache based side channel attack on ECDSA. It exploits the vulnerability present in elliptic curve cryptographic algorithms that are implemented using functions of the OpenSSL cryptographic library. In an elliptic curve E over a finite field with a point G ∈ E,the attack recovers partial bits of k(ephemeral key) used in the computation of kG during the signing process. In OpenSSL 0.9.8a using secp256k1 curve, the scalar multiplication kG is computed using *w*NAF algorithm.

The attack is implemented using two architectures i.e. spy controller architecture and spy threads architecture when the spy and victim are present in the same core .These architectures interrupt victim and makes sure that the victim runs along-with the spy threads. The slices allocated to the victim should be small enough so that it is not able to complete more than one add or double operation during its run slice. Spy threads use Flush + Reload technique to detect whether a particular cache line is accessed or not.Monitoring accesses to sets of the memory lines is one of the strength of the Flush + Reload technique. This technique is based upon fact that if data is in cache then it will take less time to load than the data which present in the main memory.

The attack was also performed when the spy and the victim are on different cores. This attack is performed on both ECDSA and DSA. Also in addition to probing single cache line, multiple cache lines are probed for better accuracy. This attack gives the double and add sequence which gives the partial information about ephemeral keys.This partial information can then be used to recover full private key using lattice attack.

Acknowledgement

We thank our guide Prof. Bernard Menezes for his constant motivation,suggestions and guidance throughout our MTP journey.We would also like to extend our gratitude towards Mr. Bholanath Roy and Mr. Ravi Prakash Giri for helping us understanding the architectures developed by them.

Table of Contents

List of Figures

List of Tables

Chapter 1

Problem Statement

1.1 Motivation

This document describes the cache based side channel attack being carried out on Elliptic Curve Digital Signature Algorithm(ECDSA). Cache based side channel attacks are focused particularly as it reveals the information of the key based on the weakness in the physical implementation of the algorithm rather than algorithm itself. Cache is a shared resource being used by multiple processes at a time. Though, actual data of the process cannot be recovered from the cache due to protection mechanisms employed, it is however possible to know the access patterns of the processes using the cache.This is the main idea for using the cache based side channel attacks for ECDSA.

1.2 Goals of MTP

- To understand the control flow of ECDSA in OpenSSL during the signing operation.
- Perform initial experiments in recovering the partial ephemeral key using FLUSH+RELOAD technique of cache based side channel attack.
- Obtain the double and add sequence generated by ECDSA during the signing operation in single-core and multi-core environment.
- Analyze the results obtained w.r.t different architectures.

Chapter 2

Background

2.1 Side Channel Attacks

A side channel attack in cryptography is an attack which is based on information obtained from the physical implementation of algorithm, rather than its theoretical weaknesses.This can be done on the basis of the information obtained from timing,power consumption, cache activity etc.[1]

Some popular types of side channel attacks are as follows[8]:

- **Timing attack :** This is the least restrictive type of attack where the attacker just measures the timing behaviour of multiple executions of the victim while generating the ciphertext.
- **Cache attack :** This attack depends on the proficiency of the attacker to monitor the cache lines accessed by the victim.
- **Power-monitoring attack :** This attack makes use of different amount of power consumed by the hardware while performing the operations.

2.2 Cache based side channel attack

Cache based side channel attacks are possible as the cache is shared among processes and there are some specific libraries that are shared by the processes e.g Openssl. Though the attacker cannot know the data of the victim, it is possible to know the cache lines which were accessed by the victim by monitoring the cache when they use a shared library. It takes the advantage of the information leakage due to time differences when data is

[1] https://en.wikipedia.org/wiki/Side-channel_attack

retrieved from the cache instead of the memory[7]. The cache is a fast and small memory in between the CPU and memory to avoid the latency in the data retrieval. Modern level cache has multiple levels to improve the efficiency in the data retrieval.There is significant difference in time when data is accessed from cache or main memory.

There are three main categories of the cache side channel attacks:

- **Time driven attack :**This is the least restrictive type of attack where the attacker just measures the timing behaviour of multiple executions of the victim while generating the ciphertext.
- **Access driven attack :**In this attack, the attacker has access to the profile of the victim's cache i.e the sequence of sets accessed by the victim[11].
- **Trace driven attack:**This is similar to the access driven attack but does not require the knowledge of the order in which the cache lines were accessed[1]. It needs to know only the cache's sets which were accessed irrespective of their order.

Such side channel attacks are mainly done in two steps:

- **Information Gathering :** In this step the attacker monitors the sets of cache accessed by the victim as coarse grained information.
- **Information Analysis :** In this step the attacker tries to extract fine-grained information from the information obtained in above step.

2.3 Techniques for cache based side channel attacks

2.3.1 FLUSH + RELOAD

The FLUSH + RELOAD attack works by splitting the time into slots. The spy flushes the monitored cache line at the start of the slot. After the slot ends, the data is loaded by the spy from the memory[17]. If the data is accessed faster it means that it was already present in the cache and thus accessed by the victim. As a result, victim's cache accesses can be monitored.

2.3.2 PRIME + PROBE

Prime + Probe is an access driven cache based side channel technique that is carried out in 3 stages[10] as under:

- **Priming stage :** The attacker fills the sets to be monitored with its own data.
- **Victim accessing stage :** The attacker waits for the victim to use the monitored cache lines
- **Probing stage :** In this stage the attacker again accesses his own data. If the time spent to access the data is less, the data was already present in the cache and the monitored cache lines were not used by the victim. However if the victim had accessed the cache lines, the data of the attacker would have been flushed out from the cache and the time to access the data would have been more.

Chapter 3

Algorithms

3.1 ECDSA

Elliptic Curve Digital Signature Algorithm(ECDSA) is a public key cryptography protocol which is a variant of Digital Signature Algorithm(DSA). The security of ECDSA lies in the computational intractability of Elliptic Curve Discrete Log Problem(ECDLP) as the Discrete Log Problem(DLP) in DSA. The ECDLP problem[16][2] states that given an elliptic curve and two points G and H over a finite field, find scalar k such that

$$H = kG$$

Elliptic Curve Cryptography(ECC) provides higher encryption strength than other algorithms that depend on hardness of DLP or the factorization of large primes. Hence ECDSA uses shorter keys than DSA for the same encryption strength.

3.1.1 Algorithm

Let E be the elliptic curve defined over finite field F_q . Generator G ∈ E is a point of large prime order n. The field size q is either a power of 2 or large odd prime.
Let a be the private key and α = aG be the public key. The calculation of private key from public key requires to solve a hard problem-ECDLP.

3.1.1.1 Signing

Suppose Alice with public-private key pair α,a wants to send message m to Bob. For signing the message, Alice should perform the following steps:

1. Compute the hash of the message, e(m).Take h to be leftmost ' bits of h (where '= min($\log_2$(q); length of hash))

2. Select a random number k $\in Z_n$
3. Calculate point (x,y)= k[G] $\in$ E
4. Calculate r = x mod n ; go to step 2, if r=0
5. Calculate $s = k^{-1}(h + ra)$; go to step 2, if s=0
6. Alice now sends (m,r,s) to Bob

3.1.1.2 Verification

To verify the signature which is sent by Alice, Bob must perform following steps:

1. Check whether the parameters are correct i.e r,s $\in Z_n$, Alice's public key is valid, $\alpha \neq 0$, $\alpha \in$ E
2. Using the hash function used during signing calculate h= hash(m) mod n
3. Calculate $s = s^{-1}$mod n
4. Calculate point (x,y) =[hs]G+[rs]Q
5. Calculate x mod n . If it equals r, accept it otherwise reject the signature

3.1.2 Vulnerability

The vulnerability of ECDSA lies in the step 2 of the signing algorithm. Improper selection of random number k can pose threat security.In the past, it lead to breaking of Sony PS3 implementation of ECDSA[16]. If the value of the ephemeral key(k) is known, the private key can be obtained. Following equation can be used to obtain the value of private key:

$$a = (sk - h)r^{-1}$$

All the values (m,r,s) can be obtained by eavesdropping and so the only unknown value is k, recovery of which can lead to recovery of private key. Hence the main intention of the attack is to recover k so that the private key of ECDSA can be obtained.

3.1.3 Implementation of Scalar Multiplication

The scalar multiplication in OpenSSL and other libraries are mostly done is one of the following two ways[16]:

- Montgomery ladder
- w-ary non-adjacent form(wNAF)

3.1.3.1 Montgomery ladder

In Montgomery ladder method double and add is done for both the set bit and zero bit of the ephemeral key to make it resilient to side channel attacks. The algorithm is as follows:

Input: Point P, k bits, scalar n
Output: Point nP
$R_0 \leftarrow \theta$
$R_1 \leftarrow P$
for i from k to 0 do
    if $n_i = 0$ then
        $R_1 \leftarrow R_1 + R_0$
        $R_0 \leftarrow 2R_0$
    else
        $R_0 \leftarrow R_1 + R_0$
        $R_1 \leftarrow 2R_1$
    end
end

Algorithm 1: Scalar multiplication using montgomery ladder

However, though both double and add functions are performed for both set and zero bit, it is still susceptible to side channel attacks. It can be found out which block of the *if* statement is executed and eventually whether the bit is set or is zero.This has been exploited in OpenSSL version 1.0.1e for sect571r1 curve [16]

3.1.3.2 w-ary non-adjacent form(wNAF)

In step 3 of ECDSA we need to calculate scalar multiplication [d]P for integer d $\in [0....2^l]$ where $l = min(log_2(q)$,bit length of the hash). The wNAF method utilize technique of preprocessing on P and the subtraction and addition in the elliptic curve group have same cost, to obtain a significant improvement in the performance[2]. So for this, first a window of size w is chosen, which is w=3 for OpenSSL's secp256k1 curve.Then $2^w - 2$ extra points are precomputed which has additional one point doubling and $2^{w-1} - 1$ point additions. The values $\{\pm G, \pm 3G..... \pm (2^w - 1)G\}$ are stored.

Then d is converted into NAF form. This can be done as mentioned in algorithm 2.According to this algorithm we can write d as $d = \sum_{i=0}^{l-1} d_i.2^i$, where

$d_i = \{\pm 1, \pm 3.... \pm (2^w - 1)\}$.This is called NAF form because for every non zero element d_i, there are at least w+1 zero values that follows[2].

Input: window width w and scalar d

Output: d in wNAF: $d_0, ..., d_{l-1}$

$l \leftarrow 0$

while $d > 0$ do

    if $dmod2 = 1$ then

        $d_l \leftarrow dmod2^{w+1}$

        if $d \geq 2^w$ then

            $d_l \leftarrow d_l - 2^{w+1}$

        end

        $d = d - d_l$

    else

        $d_l = 0$

    end

    $d = d/2$

    $l+ = 1$

end

Algorithm 2: Conversion to NAF form

From the output obtained from above algorithm, the scalar multiplication is computed using double and add.For each value, double operation is performed and for each non-zero value add operation is performed after the double operation with the precomputed values calculated initially.The algorithm is as follows:

Input: scalar d in wNAF $d_0, d_1, ..., d_{l-1}$ and precomputed points $\{G, \pm[3]G, \pm[5]G, ..., \pm[2^w - 1]G\}$

Output: $[d]G$

$Q \leftarrow \theta$

for j from l-1 down to 0 do

    $Q \leftarrow [2]Q$

    if $d_j \neq 0$ then

        $Q \leftarrow Q + [d_j]G$

    end

end

Algorithm 3: Computation of kG using *w*NAF

Example:

Let the ephemeral key,k= 793789651345567
Using algorithm 2, the key converted into *w*NAF form can be written as:
-1000050000000-1000-70000-700000-70003000-10000-700070001

The scalar multiplication is then computed using Algorithm 3 by:

- performing double function for zero digit
- performing double and add function for non-zero digit

These double and add operations are performed from LSB to MSB of the converted *w*NAF key.Before starting with these operations, $2^w - 2$ values are precomputed.Here as w=3, the precomputed values will be $\{\pm 3G, \pm 5G, \pm 7G\}$ in addition to $\pm G$. There are $2^{w-1} - 1$ additions and one doubling for precomputations as under:

G by **doubling** gives 2G
3G=G+2G (**1 addition**)
5G=3G+2G (**1 addition**)
7G=5G+2G (**1 addition**)

The corresponding negative values are obtained. As a result the double and add sequence for precomputation will be **DAAA**.As the computation is done from LSB to MSB,for convienience the *w*NAF is reversed as follows:

10007000-70000-10003000-700000-70000-7000-1000000050000-1

The corresponding double and add sequence will be:
DDDDADDDDADDDDDADDDDADDDDADDDDDDADDDDDADDDDA
DDDDADDDDADDDDDDDDADDDDDA

The original ephemeral key is then obtained as:
((G*2)*2)*2)*2)+7G)*2)*2)*2)*2)-

7G)*2)*2)*2)*2)*2)-G)*2)*2)*2)*2)+3G)*2)*2)*2)*2)-7G)*2)*2)*2)*2)*2)-
7G)*2)*2)*2)*2)-7G)*2)*2)*2)-G)*2)*2)*2)*2)-7G)*2)*2)*2)*2)-
G)*2)*2)*2)*2)*2)*2)*2)+5G)*2)*2)*2)*2)-G)
= 793789651345567G

3.1.4 ECDSA implementation in OpenSSL

For getting the ephemeral key from ECDSA, the experiments were carried out on OpenSSL 0.9.8a on the secp256k1 curve which is mostly used in Bitcoins. For signing using ECDSA, the secp256k1 curve uses the *w*NAF implementation for scalar multiplication[5].Following are the functions which is used by OpenSSL to sign using ECDSA:

Before signing, ECDSA keys are generated and curve is constructed. This curve is then associated with the keys.

EC_KEY *EC_KEY_new(void)
This function will return a new EC_KEY with no associative curve. EC_KEY represents a public key and optionally, the associated private key.
EC_GROUP *EC_GROUP_new_by_curve_name(int nid)
This method is used to construct the curve by passing the nid of the curve. For the experiments, the curve used it secp256k1 and its nid is **NID_secp256k1**. This method will create the curve if its nid is in the list, else returns error.

EC_KEY_set_group()
A curve is associated with the EC_KEY using this method. When EC_key is associated with the curve, public key - private key pair is created using EC_GROUP object.

EC_KEY_generate_key()
This method generates a new public and private key for the EC_KEY . EC_KEY must have an EC_GROUP object associated with it before a call to this function.The private key is a random integer.The public key is an EC_POINT on the curve calculated by multiplying the private key with the generator.

ECDSA_SIG* ECDSA_do_sign(const unsigned char *dgst, int dgst_len, EC_KEY *eckey)
ECDSA_do_sign() is wrapper function for **ECDSA_SIG *ECDSA_do_sign_ex(const**

unsigned char *dgst, int dlen,const BIGNUM *kinv, const BIGNUM *rp, EC_KEY *eckey) with kinv and rp set to NULL.ECDSA_do_sign_ex() computes the digital signature of hash value **dgst** with **dgst_len** bytes using the private key **eckey**.

During signature, **ECDSA_do_sign_ex()** will call **ECDSA_sign_setup()**. Inside this function, ephemeral key is generated using **BN-rand_range()** function. (i.e. the second step of ECDSA signature algorithm).

int ECDSA_sign_setup(EC_KEY *eckey, BN_CTX *ctx,BIGNUM **kinv, BIGNUM **rp)
ECDSA_sign_setup() is used to precompute parts of the signing operation. **ctx** is a pointer to **BN_CTX** structure and eckey is the private EC key. The precomputed values are returned in **rp** and **kinv** and can be used in a later call to **ECDSA_sign_ex()** or **ECDSA_do_sign_ex()**.

From the perspective of the side channel attack,step 3 of the signing algorithm i.e computation of kG is important.In order to calculate kG, **ECDSA_sign_setup()** will call **ECDSA_point_mul()** which in turn will call **ec_wNAF_mul()** to compute this scalar multiplication using *w*NAF.

int ec_wNAF_mul(const EC_GROUP *group, EC_POINT *r, const BIGNUM *scalar, size_t num, const EC_POINT *points[ ], const BIGNUM *scalars[ ], BN_CTX *ctx) This method will first convert scalar into *w*NAF form.The size of window used during conversion depends on the number of bits in scalar.

```
#define EC_window_bits_for_scalar_size(b) \
        ((size_t)\
        ((b) >= 2000 ? 6 : \
        (b) >= 800 ? 5 : \
        (b) >= 300 ? 4 : \
        (b) >= 70 ? 3 : \
        (b) >= 20 ? 2 : \
        1))
```

Then the precomputation of the values $\{\pm G, \pm[3]G, ..., \pm[2^w - 1]G\}$ is done based on window size and the ephermeral key is converted into *w*NAF using add and double

function.

To compute *w*NAF, **ec_wNAF_mul()** calls **ec_GFp_simple_add()** and **ec_GFp_simple_dbl()** functions to compute point addition and point double respectively. For zero bit, **ec_GFp_simple_dbl()** is called and for non-zero bit both **ec_GFp_simple_dbl()** and **ec_GFp_simple_add()** are called.

3.2 DSA

Digital Signature Algorithm(DSA) is a Federal Information Processing Standard i.e publicly announced standards developed by United States federal government for digital signatures.[1] The security of DSA lies in the computational intractability of Discrete Log Problem(DLP). DSA is used widely used algorithm for signing digital signatures.

3.2.1 Algorithm

The generation of the key for the algorithm is done in two parts. First, the parameters are generated and shared between users and then public key and private key are generated.

3.2.1.1 Parameter Generation

1. Select a hash function H.
2. Decide the length of the key L and N.
3. Select a prime number q of length N.
4. Select a prime number p of length L such that p-1 is multiple of q.
5. Select g whose multiplicative order modulo p is q.

The parameters of the algorithm are (p,g,q) that are shared between the users.
For every user following private and public keys are generated based on above parameters:

1. Choose a random secret key x.
2. Calculate the public key y=g^x mod p.

[1] `https://en.wikipedia.org/wiki/Digital_Signature_Algorithm`

3.2.1.2 Signing

1. Select a random number k such that 1<k<q.
2. Compute r = (g^kmod p) mod q ; if r=0, go to step 1
3. Compute s= k^{-1}(H(m) + xr); if s=0, go to step 1
4. (r,s) is the signature

3.2.1.3 Verification

1. If the conditions 0<r<q and 0<s<q are not met than the signature is dismissed
2. Compute w = s^{-1} mod q
3. Compute u_1= H(m) * w mod q
4. Compute u_2 = r *w mod q
5. Compute v = ($g^{u_1} y^{u_2}$ mod p)mod q
6. The signature is invalid if $v \neq r$

3.2.2 Vulnerability

The vulnerability of DSA lies in the step 1 of the signing algorithm. Improper selection of random number k can pose threat security. If the value of the ephemeral key(k) is known, the private key can be obtained. Following equation can be used to obtain the value of private key:

$$x = (sk - H(m))r^{-1}$$

All the values (m,r,s) can be obtained by eavesdropping and so the only unknown value is k, recovery of which can lead to recovery of private key. Hence the main intention of the attack is to recover k so that the private key of DSA can be obtained.

3.2.3 Implementation of Scalar Multiplication

The scalar multiplication for DSA in OpenSSL and other libraries is mostly done using Sliding Window Exponentiation(SWE). SWE is widely used as it reduces the average number of multiplications needed for signing and thus reducing the overall time for the signing operation[12].

3.2.3.1 Sliding Window Exponentiation

The exponent e is divided into a set of windows e_i. The window e_i can be a set of zeros or a non-zero window with the start and end of the window as 1. If the size of window is w, the length of the non-zero window can be anything between 1 and w. Thus the value of the non-zero window can be any number between 1 and 2^w-1. OpenSSL uses window size w=4 and it stores the precomputations for faster operations. The precomputations b^1,b^3,b^5,..., b^{15} are stored in an array as g[0],g[1],..,g[7].Following algorithm is used for the scalar multiplication[12]:

Input: N bit exponent e represented as n windows e_i , length of the windows l(e_i),window size w, base b, modulo m
Output: b^e mod m
g[0] ←b mod m
p ←MULT(g[0],g[0]) mod m
for j from 1 to $2^{w-1} - 1$ do
    g[j] ← MULT(g[j-1],p) mod m;
r ←1;
for x ← n to 1 do
    for y ← 1 to l(e_i) do
        r ← MULT (r,r) mod m;
    if $e_i \neq 0$ then r ← MULT(r,g[(e_i)/2]) mod m;
return r;

Example: Let k = 14264
The binary representation of k is 11011110111000
The algorithm divides the k in the window of <=4 when there are non-zeros at the start and the end as that is the default size of OpenSSL when the key is 160 bits.The window of all zeros can be of any length. So the k is divided into the windows as follows:

11 | 0 | 1111 | 0 | 111 | 000

For each bit, a sqare operation is performed and for each window containing non-zero bits, a multiply operation is performed at the end of the window. Therefore, the operations performed for the above value of k are as under:

Window	11	0	1111	0	111	000
Operations	SSM	S	SSSSM	S	SSSM	SSS
Multiplier for M	a^3	-	a^{15}	-	a^7	-

The above operations evaluates to the following result:

$1 \rightarrow 1 \rightarrow 1 \rightarrow a^3$	(SSM)
$a^3 \rightarrow a^6$	(S)
$a^6 \rightarrow a^{12} \rightarrow a^{24} \rightarrow a^{48} \rightarrow a^{96} \rightarrow a^{111}$	(SSSSM)
$a^{111} \rightarrow a^{222}$	(S)
$a^{222} \rightarrow a^{444} \rightarrow a^{888} \rightarrow a^{1776} \rightarrow a^{1783}$	(SSSM)
$a^{1783} \rightarrow a^{3566} \rightarrow a^{7132} \rightarrow a^{14264}$	(SSS)

3.2.4 DSA implementation in OpenSSL

The signing operation in DSA starts with the **DSA_sign()** which is present in crypto/dsa/dsa_sign.c . **DSA_sign()** then calls **dsa_do_sign()** in dsa_ossl.c which in turn calls **dsa_sign_setup()**. The signing operation is done in **dsa_sign_setup()**.

dsa_sign_setup() calls the function **DSA_BN_MOD_EXP()** which in turn calls **BN_mod_exp_mont()** where the exponentiation is computed. **BN_mod_exp_mont()** is present in crypto/bn/bn_exp.c . **BN_mod_exp_mont()** calls **BN_mod_mul_montgomery()** which calls **BN_sqr()** to perform squaring operation and **BN_mul()** to perform multiply operation.

In the FLUSH + RELOAD technique, **BN_sqr()** and **BN_mul()** functions are probed for the side channel attack.

Chapter 4

Experimental models

We recovered partial ephemeral key from ECDSA using two architectures:

1. Spy Controller Architecture
2. Spy Threads Architecture

The spy controller architecture was developed by Mr. Bholanath Roy for side channel attack on DSA and spy threads architecture was developed by Mr. Ravi Prakash Giri for side channel attack on AES. We used these architectures and modified it according to the requirements for ECDSA.

4.1 Spy Controller Architecture

In this architecture we have a spy controller running on one core and a spy ring(a group of threads running in round robin fashion) on another core[15]. These spy threads will be interleaved with that of the victim which is performing the ECDSA operation and whose ephemeral key is to be found out.These threads are executed in a way that the victim is preempted after some limited operations. Then it should be determined if the victim has brought some data into the cache lines which in turn reveals some information about the ephemeral key. The spy threads are executed one after another with victim running between execution of any two threads. The victim is preempted in short duration due to the Completely Fair Scheduler(CFS). If the spy threads are not present, the victim can run on the processor for long time and perform the signing process but by introducing the spy threads we are exploiting the CFS.

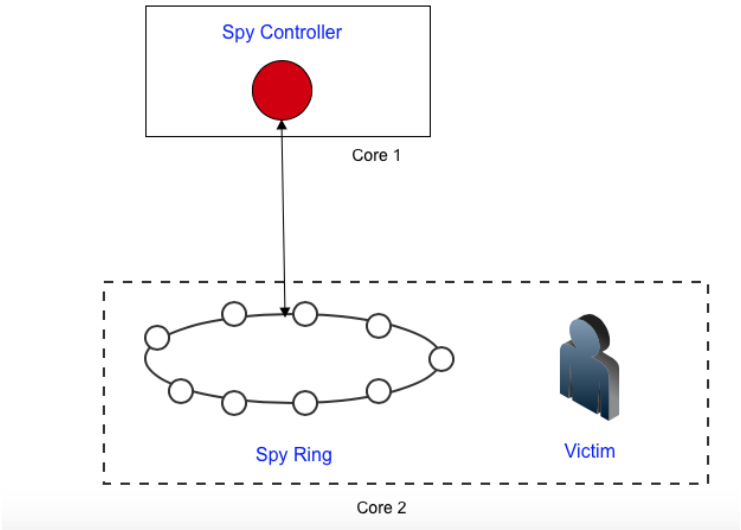


Figure 4.1: Spy Controller Architecture

The CFS is employed in many Linux versions which keeps track of virtual runtimes to make sure that the overall CPU times allocated to all processes and threads are nearly equal. So the total time given to victim(V) is equal to the times given to each spy thread.

Suppose there are *n* threads and each thread is running for *x* duration. Due to CFS, victim will also be executed for x duration by the time all the threads complete their execution.

overall CPU running time = time taken by n spy threads to run + time taken by victim to execute

As the victim is interrupted after some time by the threads, the victim runs for x/n time in each uninterrupted run. As a result, the greater the value of n, the victim will be executed for shorter duration. The shorter the duration, lesser operations are performed by the victim and better results are obtained.

4.1.1 Working of spy threads and spy controller

Each spy threads will probe the cache lines of **ec_GFp_simple_dbl()** and **ec_GFp_simple_add()** functions to get information whether victim has accessed these functions or not.

The spy threads will perform following things whenever it is scheduled by the CPU:

- **FLUSH :** The spy thread will flush the cache lines of the double and add functions. For flushing the cache lines **clflush** instruction is used.
- **WAIT :** It will wait for the victim to compute double and add functions and bring the functions into the cache
- **RELOAD :** The spy will then try to access the double and add functions and it measures time to access these operations. If the access time is higher than certain threshold then victim has not accessed these functions as the cache lines were flushed and the functions were not present in the cache but were accessed from the main memory. However, if the access time is less than certain threshold, it can be concluded that these functions were used by the victim.

Spy_Thread(i)

```
while(true)
    wait (cond)
    for cacheLine in add function
        if(accessTime[cacheLine]<THRESHOLD)
            isAccessed[cacheLine] = true
            clflush(cacheLine)
    for cacheLine in dbl function
        if(accessTime[cacheLine]<THRESHOLD)
            isAccessed[cacheLine] = true
            clflush(cacheLine)
    Finish = true
    delay loop // time=δ
```

Algorithm 3: Working of Spy threads in Spy controller architecture

When a spy thread completes its execution, it will signal spy controller using a shared variable *Finish*. Spy controller continuously loops until *Finish* variable is set to true. This denotes the end of execution of one thread and then it waits for δ time before waking up another spy thread which waited the longest.

Spy Controller

```
while (true)
    while(Finish ≠ true);
    delay loop // time=δ
```

signal(nextThread)
finished = false

Algorithm 4 : Working of Spy controller

4.1.2 Thread synchronization

In order to have synchronization among threads, mutex variables and conditional variables are used.

Mutex variables acts like a lock when shared variables are accessed (Current-Thread,Finish). In mutex only one thread can enter the block of code protected by mutex variable i.e it locks the code after entering[13]. If several threads try to enter the code protected by mutex variable only one will be successful. Other threads will be allowed to access that code only after the one using it unlocks it. Threads will "take turns" in accessing that code.

```
pthread_mutex_lock( mptr[tid]); // lock on mutex variable
    while (*currThread != tid) //shared data : currThread
    {
        pthread_cond_wait(cvptr[tid], mptr[tid]); //waiting on conditional variable
    }
pthread_mutex_unlock(mptr[tid]);

pthread_cond_signal( cvptr[*currThread] ); // sends signal to next spy thread
```

Code snippet for thread synchronization

Conditional variables are another way for synchronization of threads. As mutex variable synchronize the threads based on the access to shared data, similarly conditional variables synchronize threads on the basis of the value of data. It is used to keep thread in blocking state until certain condition is met. Without condition variables,the threads need to be polled continuously (possibly in a critical section), to check if the condition is satisfied. As the thread would be continuously busy, it can be very resource consuming. A condition variable achieves the same goal without polling. It is always used in along-with with a mutex lock.

4.2 Spy Threads architecture

This architecture performs attack on ECDSA using only spy threads unlike previous architecture that used spy controller for synchronization among threads. In this architecture there is only spy ring(group of spy threads) that runs with the victim on the same core. The basic idea of the attack is the same i.e. to preempt victim at short durations to know the activities performed by the victim. The victim will run for x/n time in single run where x is the total duration of spy and n is the number of threads used in the spy ring.

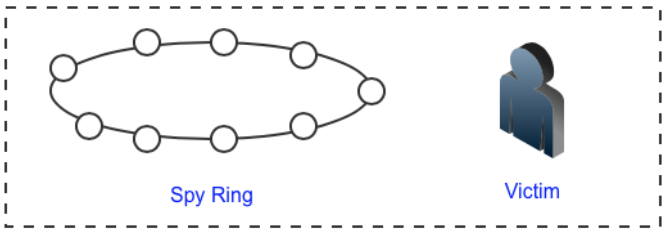


Figure 4.2: Spy Threads Architecture

The working of the spy threads remains the same as specified in 4.1.1 with difference in waking up the next thread that is blocked.

Signal Handler

```
function V(Semaphore S, Integer I)
    sem_post(s[I+1])
```

Spy Threads

```
timer_create(clock id, sigevent, timerid)
function P(Semaphore S, Integer I)
    sem_wait(s[I])
    for cacheLine in add function
        if(accessTime[cacheLine]<THRESHOLD)
            isAccessed[cacheLine] = true
        clflush(cacheLine)

    for cacheLine in dbl function
        if(accessTime[cacheLine]<THRESHOLD)
            isAccessed[cacheLine] = true
```

clflush(cacheLine)

newTimerValue ← t
timer_settime(timerid, NULL, newTimerValue,oldTimerValue)

Algorithm 5 : Working of Spy threads

4.2.1 Thread Synchronization

In this architecture timers are used for synchronization among threads as opposed to mutex variables in earlier architecture. When a thread completes its execution it will start a timer and control goes from user mode to kernel mode. When timer expires the kernel will send a signal that will call the signal handler[9]. The signal handler will then signal the next thread to continue its execution. So the victim will execute in the between these threads before the next thread starts executing.

timer_create(CLOCKID, &sev, &timerid); // Creating a timer which will be used by the threads.

its.it_value.tv_sec = 0;
its.it_value.tv_nsec = 1000;
its.it_interval.tv_sec = 0;
its.it_interval.tv_nsec = 0;

ret = timer_settime(timerid, 0, &its, NULL); // Arming the timer with the specified time.

Working of timers

int timer_create(clockid_t clockid, struct sigevent *sevp,timer_t *timerid) timer_create() creates a new per-process interval timer. The ID of the this timer is returned in the buffer pointed to by timerid, which should be a non-null pointer.

int timer_settime(timer_t timerid, int flags,const struct itimerspec *new_value, struct itimerspec *old_value)
timer_settime() disarms or arms the timer identified by timerid. The new_value argument is pointer to an *itimerspec* structure that specifies the new initial value and the new interval for the timer. The *itimerspec* structure is defined as follows:

```
struct timespec {
        time_t tv_sec; /* Seconds */
        long tv_nsec; /* Nanoseconds */
        };

struct itimerspec {
        struct timespec it_interval; /* Timer interval */
        struct timespec it_value; /* Initial expiration */
        };
```

it_val specifies will specify the time in nanoseconds in our case. it_interval is used to create an interval timer which is gonna expire after the specified time mentioned in it again and again. Zero means it'll expire only once.

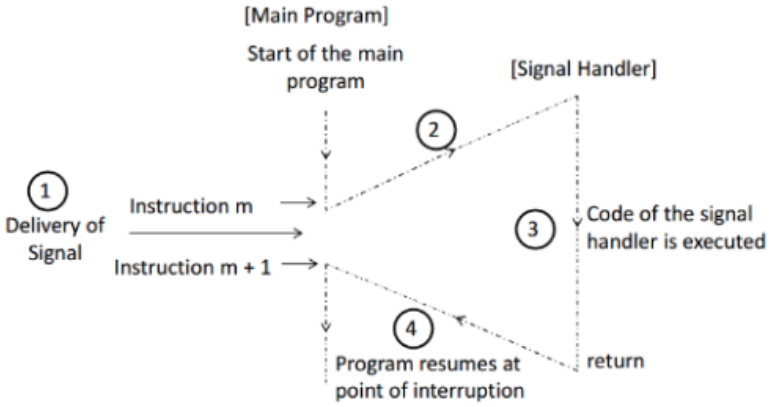


Figure 4.3: Working of signal handler

When an event occurs(such as timer expires) it will send a signal from main program to signal handler as shown figure 4.3. After receiving the signal in step 2, handler will execute its code (in this case signal handler will awake next thread to execute). When signal handler completes its execution, the execution control will be returned to the next instruction of the calling program.

In order to restrict different threads from accessing the shared variable, semaphores are used .The two kinds of semaphores are counting semaphores and binary semaphores. Here binary semaphores are used such that if semaphore value 0 means lock or block state and semaphore 1 value denotes unlock state. Initially, all thread' s semaphore value is set to 0 except one thread that will be allowed to execute. When a thread wants to

access a shared variable it calls **sem_wait()**.Thread which has semaphore value equal to 1 will be allowed to use the shared code and it's semaphore value will be set to 0. All other threads are blocked here as their semaphore value is 0. Once thread completes then it will send signal to next thread using signal handler of timer expire event. This signal handler will increment next thread semaphore value using **sem_post()**, causing the execution of next thread.

Chapter 5

Experimental Setup

5.1 Attack Scenario

A victim having a fixed private key - public key pair (p,Q where Q=pG , G being the point on the curve) is signing same message multiple times with ECDSA using **secp256k1** curve. A spy and victim are executed on the same core and also on different core. In both cases spy will monitor the cache accesses made by the victim and determine whether the victim has performed the double and add function required by ECDSA during scalar multiplication.By monitoring this, the spy can get the double and add sequence which in turn reveals the partial ephemeral key. Full recovery of the ephemeral key will require post processing using the lattice attack which is not in the scope of this document. After recovery of the ephemeral key, private key can be obtained as mentioned in section 3.2.2

5.2 Assumptions

- Victim and spy are using shared OpenSSL library, so that spy is able to determine the cache accesses made by the victim by probing the same function ans is able to flush and reload it.
- No other processes running OpenSSL is executing in the same processor as victim and spy during attack in case of single core side channel attack.
- No other processes running OpenSSL is executing in the any other processor during attack in case of multi-core side channel attack.
- In case of multi-core side channel attack memory deduplication is enable.

5.3 Steps for implementation of attack

1. First the victim is started such that it will compute its ECDSA parameters required for the signing operation.

2. The spy will then be executed and it will create n threads (starting from thread id 0 to n-1). As each thread will update the shared data values, only one thread will be allowed to be executed at a time and other threads will be blocked.

3. When initial thread will come it will bring the add and double function in cache and then it will flush the cache lines containing these function. After doing this it waits for some time and then will set shared flag to 1.

4. When victim will start its signing process, then during signing process when it performs step 3 i.e. computation of kG, it will perform some add and double operation as discussed earlier. When add and double operations are performed victim will these functions in cache.

5. When next time spy is scheduled, it can see whether victim has accessed add and double function. By probing the cache lines, spy can measure the time taken to access the cache line of add and double function and if time taken is less than a threshold it indicates that add or double function is already in cache and they are accessed by the victim.

6. If victim is getting enough time then it can perform multiple add and double operations in single run. In such case the spy cannot determine exact sequence of operations performed by the victim.So it should be ensured by the spy that this does not happen.

7. So,in case of single core attack we need to stop the victim after each operation(add or double) or in between single operation multiple time so that spy is able to get information about every operation and hence it can generate exact sequence of add and double.

8. Using the two spy-architectures that was discussed earlier, victim is given very small time such that it can only perform one operation during a single run in single core attack.

9. Similarly in case of multi-core, in order to get information about each operation performed by the victim, spy should run such that between two probe operation it should execute less than time require to perform a single operation by victim.

5.4 Attack on single core

5.4.1 Single line probing

The experiments for single line probing were done by probing the base address of the double and add function.These experiments were performed for both - the spy controller architecture and spy thread architecture. Following image is the snippet of the file obtained when these experiments were performed:

Counter	Thread	operation	Double	Add	
20,	110,	21,	0,	1	(A)
20,	111,	21,	0,	0	
20,	112,	22,	1,	0	(D)
20,	113,	22,	0,	0	
20,	114,	23,	1,	0	(D)
20,	115,	24,	1,	0	(D)
20,	116,	24,	0,	0	
20,	117,	25,	1,	0	(D)
20,	118,	26,	0,	0	
20,	119,	26,	0,	1	(A)
20,	120,	27,	1,	0	(D)
20,	121,	27,	0,	0	
20,	122,	28,	1,	0	(D)
20,	123,	29,	1,	0	(D)
20,	124,	29,	0,	0	
20,	125,	30,	1,	0	(A)
20,	126,	30,	0,	0	
20,	127,	31,	0,	1	(A)
20,	128,	31,	0,	0	
20,	129,	32,	1,	0	(D)
20,	130,	33,	0,	0	
20,	131,	33,	1,	0	(D)
20,	132,	34,	1,	0	(D)
20,	133,	34,	0,	0	
20,	134,	35,	1,	0	(D)
20,	135,	36,	1,	0	(D)
20,	136,	36,	0,	0	
20,	137,	37,	1,	0	(D)
20,	138,	37,	0,	0	
20,	139,	38,	0,	1	(A)
20,	140,	39,	1,	0	(D)

Figure 5.1: Generation of double and add sequence in single line probing

In figure 5.1, *Counter* indicates the iteration of the thread mentioned in *Thread* , *operation* specifies the which operation is being performed by victim to ensure that multiple

operations are not performed by victim,1 in *Double* specifies that Double operation was performed by victim. Similarly for *Addition*.

Here, repeating values in *operation* indicates that the victim was preempted more than once when it was performing a single operation. If the value is different, this indicates that a new operation is performed.

5.4.2 Multiple line probing

To improve the accuracy, in addition to single line probing we performed experiments that probes multiple cache lines and hence reduces the probability of an operation being missed.

Following image is the snippet of the file obatined when multiline probing is done:

Counter	Operation	Thread	Double	Add	
31,	7,	123,	0, 0, 0, 0,	0, 0, 0, 0	
31,	7,	124,	1, 1, 0, 0,	0, 0, 0, 0	
31,	7,	125,	0, 0, 1, 1,	0, 0, 0, 0	(Double)
31,	7,	126,	0, 0, 1, 1,	0, 0, 0, 0	
31,	7,	127,	0, 0, 0, 1,	0, 0, 0, 0	
31,	7,	128,	0, 0, 0, 0,	0, 0, 0, 0	
31,	7,	129,	0, 0, 0, 0,	0, 0, 0, 0	
31,	7,	130,	0, 0, 0, 0,	0, 0, 0, 0	
31,	7,	131,	0, 0, 0, 0,	0, 0, 0, 0	
31,	7,	132,	0, 0, 0, 0,	0, 0, 0, 0	
31,	8,	133,	0, 0, 0, 0,	0, 0, 0, 0	
31,	8,	134,	0, 0, 0, 0,	1, 0, 0, 0	
31,	8,	135,	0, 0, 0, 0,	0, 1, 1, 0	
31,	8,	136,	0, 0, 0, 0,	0, 0, 1, 0	(Add)
31,	8,	137,	0, 0, 0, 0,	0, 0, 1, 0	
31,	8,	138,	0, 0, 0, 0,	0, 1, 0, 1	
31,	8,	139,	0, 0, 0, 0,	0, 0, 0, 1	
31,	8,	140,	0, 0, 0, 0,	0, 0, 0, 0	
31,	8,	141,	0, 0, 0, 0,	0, 0, 0, 0	
31,	8,	142,	0, 0, 0, 0,	0, 0, 0, 0	
31,	8,	143,	0, 0, 0, 0,	0, 0, 0, 0	
31,	8,	144,	0, 0, 0, 0,	0, 0, 0, 0	
31,	8,	145,	0, 0, 0, 0,	0, 0, 0, 0	
31,	9,	146,	0, 0, 0, 0,	0, 0, 0, 0	
31,	9,	147,	1, 1, 1, 0,	0, 0, 0, 0	
31,	9,	148,	0, 0, 1, 1,	0, 0, 0, 0	(Double)
31,	9,	149,	0, 0, 1, 1,	0, 0, 0, 0	
31,	9,	150,	0, 0, 0, 1,	0, 0, 0, 0	
31,	9,	151,	0, 0, 0, 0,	0, 0, 0, 0	

Figure 5.2: Generation of double and add sequence in multiline probing

For this, we first found the total number of cache lines that will be occupied by double and add functions in case of ECDSA and square and multiply functions in case of DSA. This was done by using gdb.

Following are the number of cache lines occupied by double and add functions for ECDSA:

Table 5.1: Number of cache lines occupied by double and add functions

	Number of cache lines
Add	31
Double	24

Following is the code snippet that probes 4 lines:

```
d = ((char *)(&ec_GFp_simple_dbl));          //cache line 1
d1 = d + 67;                                 //cache line 2
d2= d + 360;                                 //cache line 6
d3 = d + 406;                                //cache line 7

a = ((char *)(&ec_GFp_simple_add)) +19;      //cache line 1
a1 = a + 325;                                //cache line 6
a2= a + 772;                                 //cache line 13
a3 = a + 990;                                //cache line 16
```

Following are the number of cache lines occupied by square and multiply functions for DSA:

Table 5.2: Number of cache lines occupied by double and add functions

	Number of cache lines
Multiply	17
Square	12

Following is the code snippet that probes 4 lines:

```
s = ((char *)(&BN_sqr)) +19;                 //cache line 1
s1 = s + 325;                                //cache line 6
s2= s + 605;                                 //cache line 10
s3=(char *)(&bn_sqr_recursive);              //a function called at the end of BN_sqr()
```

```
m = ((char *)(&BN_mul));                    //cache line 1
m1 = m + 310;;                              //cache line 5
m2 = m + 894 ;                               //cache line 14
m3=(char *)(&bn_mul_recursive);              //a function called at the end of BN_mul()
```

The experiments on multi-line probing was only performed on the spy controller architecture as the spy controller architecture was able to preempt the victim multiple times in a single operation.
In spy threads architecture, no matter how small the duration between the threads was set, the spy was not able to preempt the victim multiple times and hence could not utilize the advantage of probing multiple lines. The reason for this is, spy thread uses the timer that need to involve kernel. Kernel will send signal to run a handler for timer expiration. This involves the switch from user mode to kernel mode and then send signal to handler and then again switch to kernel mode to user mode. So this gives enough time to victim and spy is not able to interrupt the victim multiple time during a single operation.

5.5 Attack on multiple cores

In the earlier experiments, it was assumed that the spy and victim reside on the same core and hence we could able to exploit Completely Fair Scheduler(CFS). But in a more practical scenario, the victim and the spy can be on different cores. In this case, we cannot use the architectures specified in section 4.1 and 4.2.

So to attack in the case when victim and spy resides on different cores, we use the FLUSH + RELOAD technique on Last Level Cache(LLC). The LLC is shared among multiple cores while L1 and L2 cache are private to the core. Here also, the victim and spy should use a shared library like OpenSSL .In this attack, the spy is only a single program that continuously flushes the cache lines to be probed and then wait operation to perform and detects whether the victim has accessed the cache lines and thus whether the operation is performed. As earlier, this attack was also done by single line probing and multi-line probing.

The single line probing and multi-line probing is exactly similar to sections 5.4.1 and 5.4.2.

Chapter 6

Experimental Results

6.1 ECDSA

6.1.1 Add and Double operations

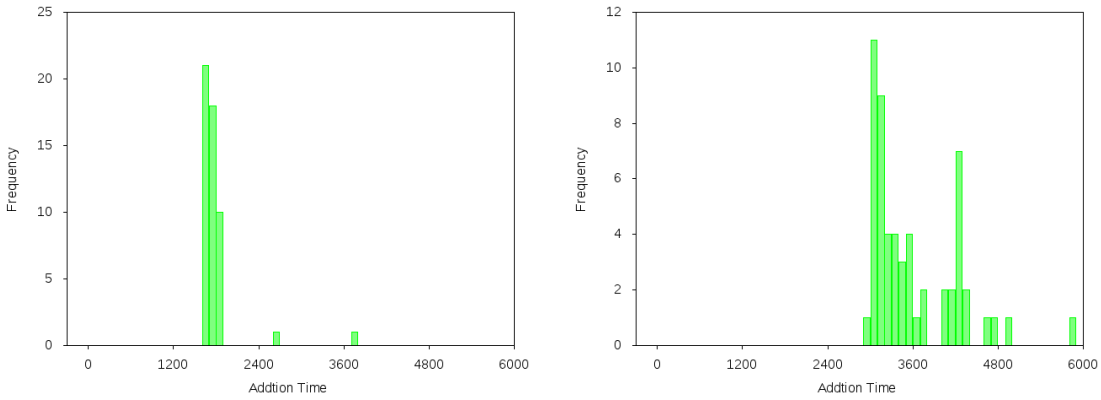
As mentioned in 5.1 we need to obtain the add and double sequence and hence the time required for double and add operation on i5 processor was computed experimentally without spy and in presence of spy. The average time taken by these functions to execute is shown in below table:

Table 6.1: Average time(in ns) for double and add operation

	Add(in ns)	Double(in ns)
Without Spy	1964	1660
With Spy	3663	3431

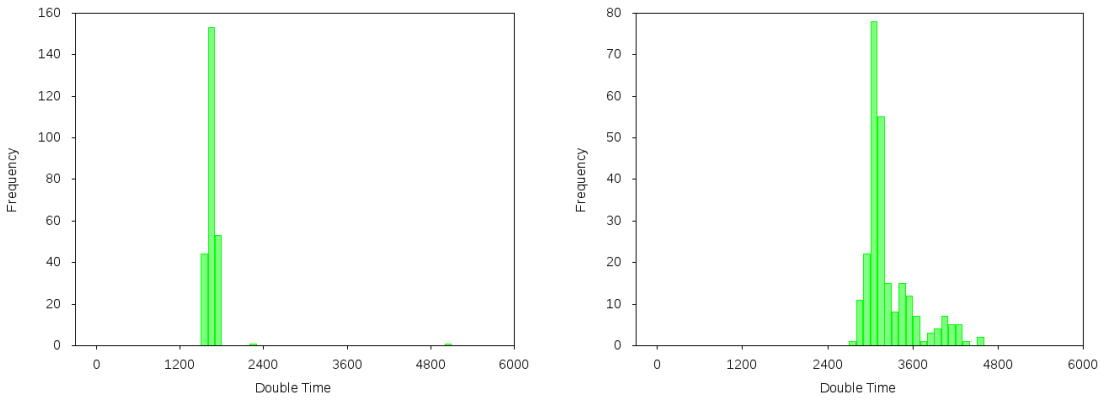
It can be observed that the time required to compute add operation is much higher in presence of spy as compared to when only victim performs the add operation. This is because the spy threads flush the cache every-time they get scheduled. As a result the victim has to access the add function from the main memory resulting in increase in the access time.

Similarly, as shown below in graphs 6.2a and 6.2b, it can be observed that the time required to compute dbl operation is much higher in presence of spy as compared to when only victim performs the dbl operation. The reason for this is same as specified above.



(a) Without spy (b) With spy

Figure 6.1: Time required for add operation



(a) Without spy (b) With spy

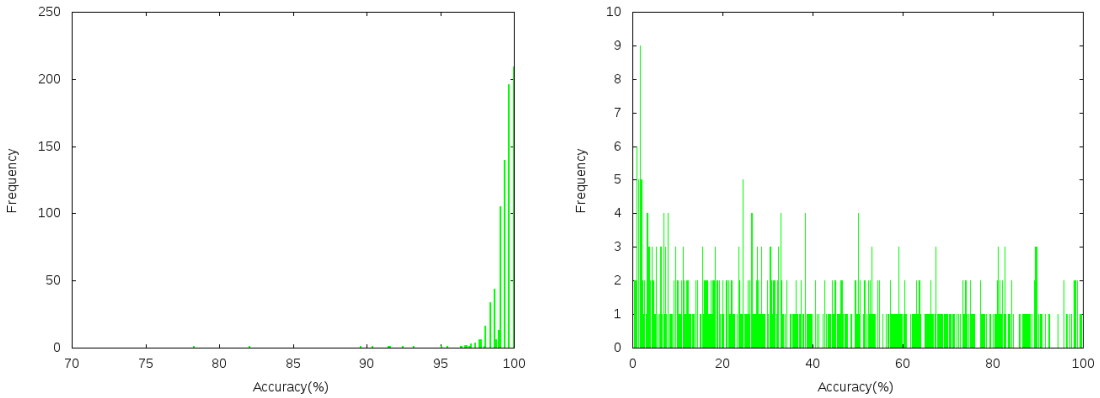
Figure 6.2: Time required for double operation

6.1.2 Attack on single core

The attack was performed on 64bit i3/i5 Intel processor running Debian GNU/Linux 8 (jessie) with OpenSSL version 0.9.8a. Number of threads were set to 150 and each thread was made to run 1000 times.

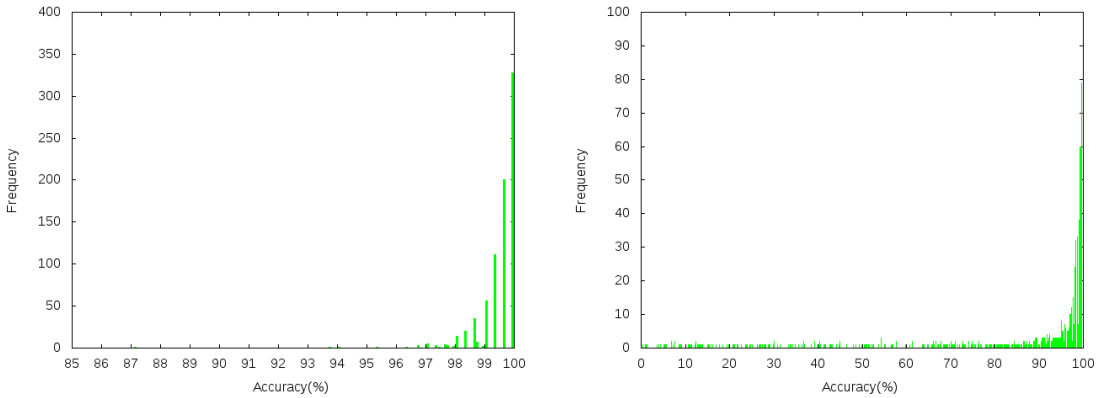
For the architecture specified in both 4.1 and 4.2 the timer value was set to 1000ns.

6.1.2.1 Single line probing



(a) Without Background Processes (b) With Background Processes

Figure 6.3: Accuracy in i3 processor for Spy Controller Architecture

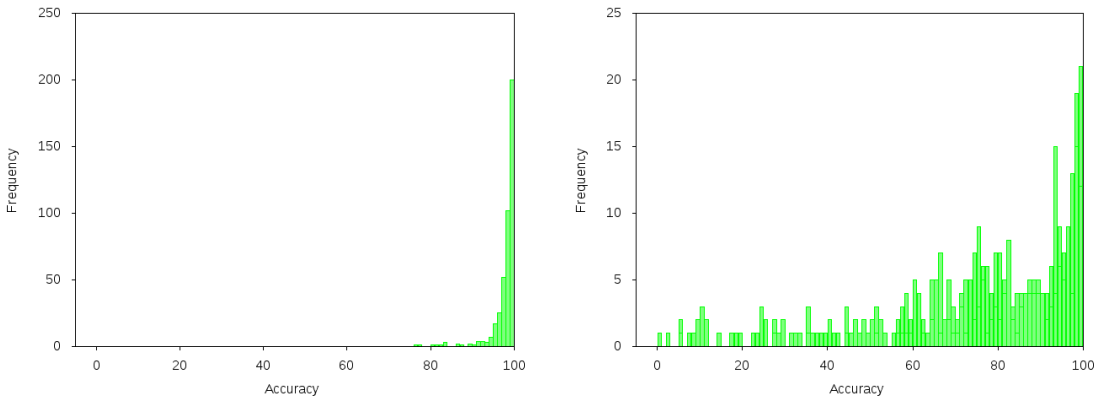


(a) Without Background Processes (b) With Background Processes

Figure 6.4: Accuracy in i5 processor for Spy Controller Architecture

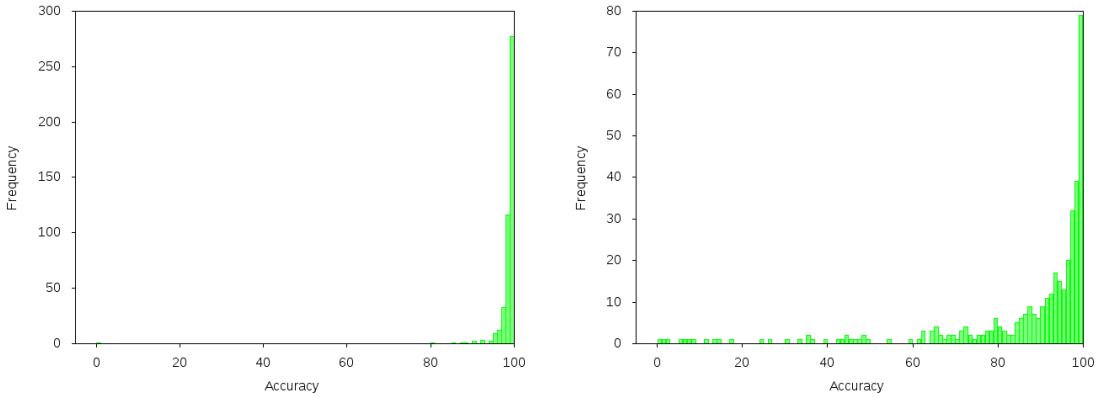
Figures 6.3a and 6.3b shows the distribution of accuracy using the spy controller architecture mentioned in section 4.1. Graph 6.3a shows the accuracy obtained in detecting the double and add sequence on i3 processor when only victim and spy are running in the system.The average accuracy obtained in this case is 99.251 %.Graph 6.3b shows the accuracy obtained in detecting the double and add sequence on i3 processor in presence of background processes like a process running 10000 loops with print statements on the same core as victim and spy. Some other random processes were also during the attack. This is to ensure that some other process is also running on the same core along-with spy and victim. The average accuracy obtained in this case is 40.337%. In this case out of 800 signatures, 219 signatures were not detected at all as there were too many background processes and as only 2 cores are present in i3 processor, the spy might not have been scheduled at all when the victim performed the signature. Figures 6.4a and 6.4b

shows the distribution of accuracy in i5 processor.Graph 6.4a shows the accuracy when only victim and spy are running in the system.The average accuracy obtained in this case is 99.49%. Graph 6.3b shows the accuracy in presence of background processes.he average accuracy obtained in this case is 83.9161%. The accuracy in this case did not drop that much as in i3 as there are 4 cores and the processes are distributed among 4 cores as opposed to 2.



(a) Without Background Processes (b) With Background Processes

Figure 6.5: Accuracy in i3 processor for Spy Threads Architecture



(a) Without Background Processes (b) With Background Processes

Figure 6.6: Accuracy in i5 processor for Spy Threads Architecture

Figures 6.5a and 6.5b shows the distribution of accuracy using the spy threads architecture mentioned in section 4.2. Graph 6.5a shows the accuracy obtained in detecting the double and add sequence on i3 processor when only victim and spy are running in the system.The average accuracy obtained in this case is 98.153 %.Graph 6.5b shows the accuracy obtained in detecting the double and add sequence on i3 processor in presence of background processes like a process running 10000 loops with print statements on the same core as victim and spy.Some other random processes were also during the attack.

This is to ensure that some other process is also running on the same core along-with spy and victim.The average accuracy obtained in this case is 76.4755%. Similarly figures 6.6a and 6.6b shows the distribution of accuracy in i5 processor.Graph 6.6a shows the accuracy when only victim and spy are running in the system.The average accuracy obtained in this case is 98.5404 %. Graph 6.5b shows the accuracy in presence of background processes. The average accuracy obtained in this case is 87.9723%. The reason for difference in accuracy of i3 and i5 in presence of background processes is same as explained above.

In short, following results are obtained for average accuracy from the above experiments:

Table 6.2: Average accuracy(%) obtained using Spy Controller and Spy Threads Architecture

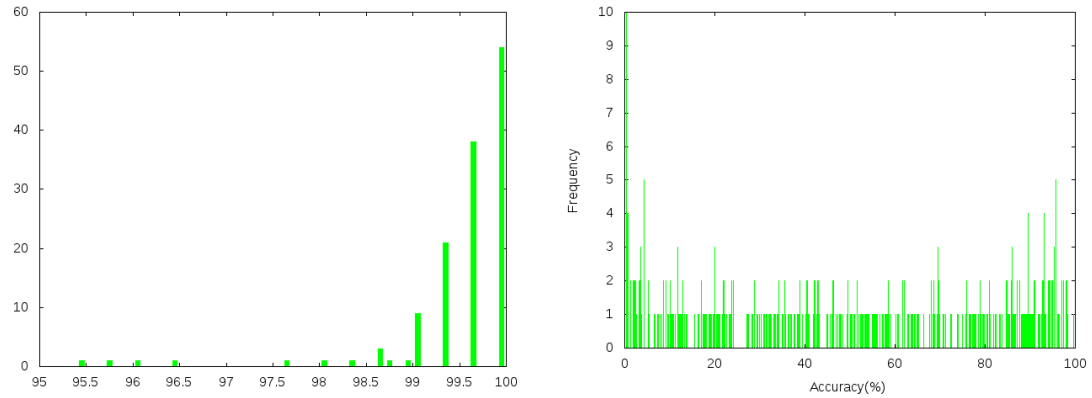
	i3 Processor		i5 Processor	
	Without Background Processes	**With Background Processes**	**Without Background Processes**	**With Background Processes**
Spy Controller Architecture	99.251	40.337	99.49	83.9161
Spy Threads Architecture	98.153	76.4755	98.5404	87.9723

From above results, it can be observed that better accuracy is obtained using the spy controller architecture in comparison to the spy threads architecture. This is because in the spy controller architecture we were able to preempt the victim multiple times than in spy threads architecture and hence the individual operations were more clearly detected. Also in spy controller architecture when there are no background processes, $\sim 41\%$ cases, perfect double and add sequence was obtained as opposed to spy thread architecture in which $\sim 9\%$ cases perfect sequence was obtained. Similar observations were also made in i3 processor.

6.1.2.2 Multi-line probing

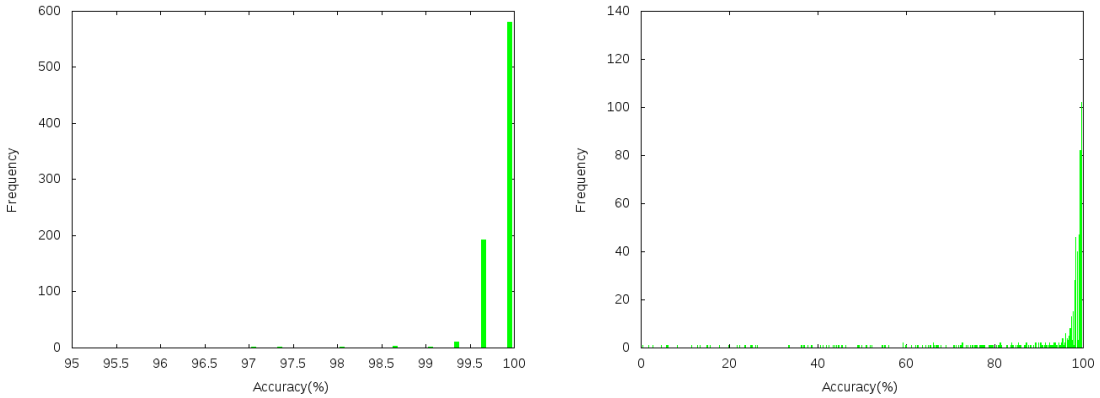
Multiline probing is done in order to improve accuracy. As already discuss in 5.4.2 we can use spy controller architecture to interrupt the victim more number of time.

Here are the results obtained on i3 and i5 by probing multiple cache line of add and double function. As we can see there is not much difference between average accuracy obtained from single line probing and multi-line probing. But we have got more number of perfect sequence by doing multi-line probing.



(a) Without Background Processes (b) With Background Processes

Figure 6.7: Accuracy in i3 processor using multi-line probing in single core environment



(a) Without Background Processes (b) With Background Processes

Figure 6.8: Accuracy in i5 processor using multi-line probing in single core environment

Following table summarizes the accuracy obtained in case of probing multiple lines in single core.

Table 6.3: Average accuracy obtained in i3 and i5 by probing multiple lines in single core

Processor	Without background processes	With background processes
i3	99.435	50.25
i5	99.888	92.013

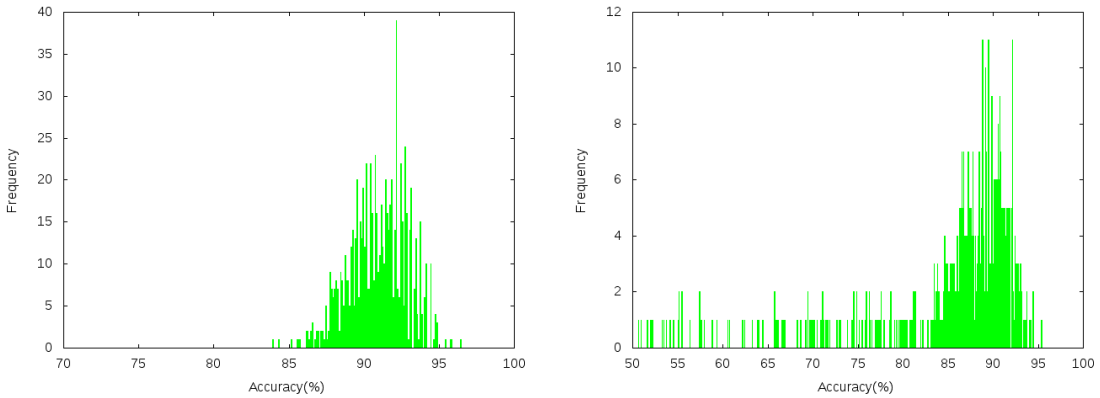
Table 6.4: % of perfect sequence obtained by single line and multi-line probing on i3 and i5 processors

Processor	Single line probing	multi-line probing
i3	~ 26%	~ 46%
i5	~ 41%	~ 72%

Here as shown in above table with multi-line probing we are getting more perfect match. Also in case of i5 we are getting better results than i3. The reason for this is, i3 system clock was slow comparison to i5 , hence the ones detecting add and double operations are more clustered in case of i5 processor. So in case of i5 system it will be more easy to detect the individual operation in comparison to i3.

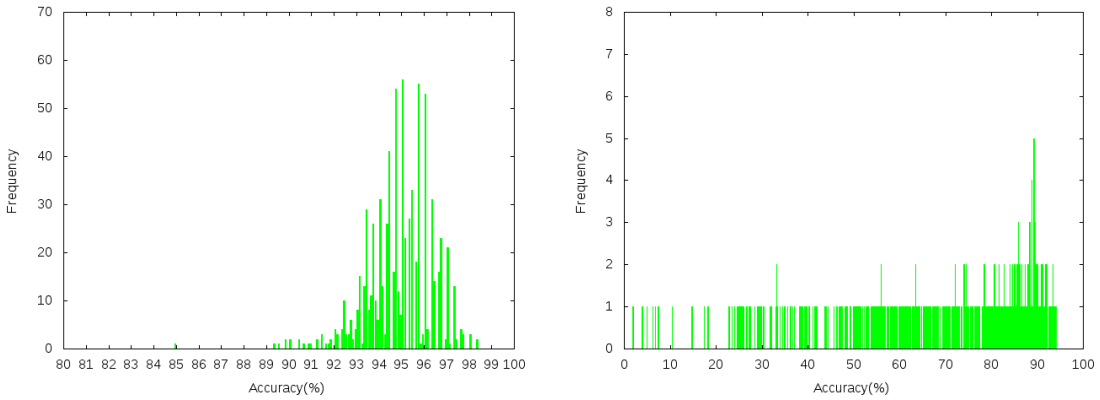
6.2 Attack on multiple cores

6.2.1 Single line probing



(a) Without Background Processes (b) With Background Processes

Figure 6.9: Accuracy in i3 processor using single-line probing in multi core environment



(a) Without Background Processes (b) With Background Processes

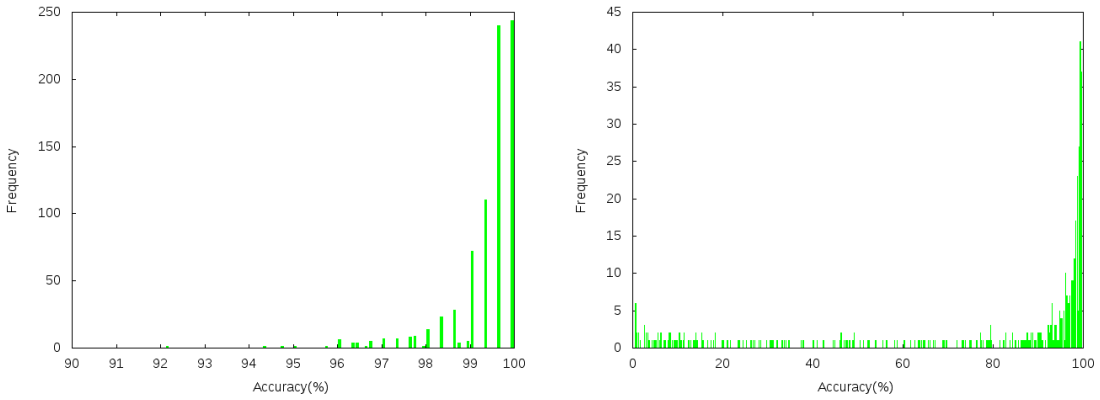
Figure 6.10: Accuracy in i5 processor using single-line probing in multi-core environment

Following table summarizes the accuracy obtained in case of probing single line in multi-core.

Table 6.5: Average accuracy obtained in i3 and i5 by probing single line in multi-core

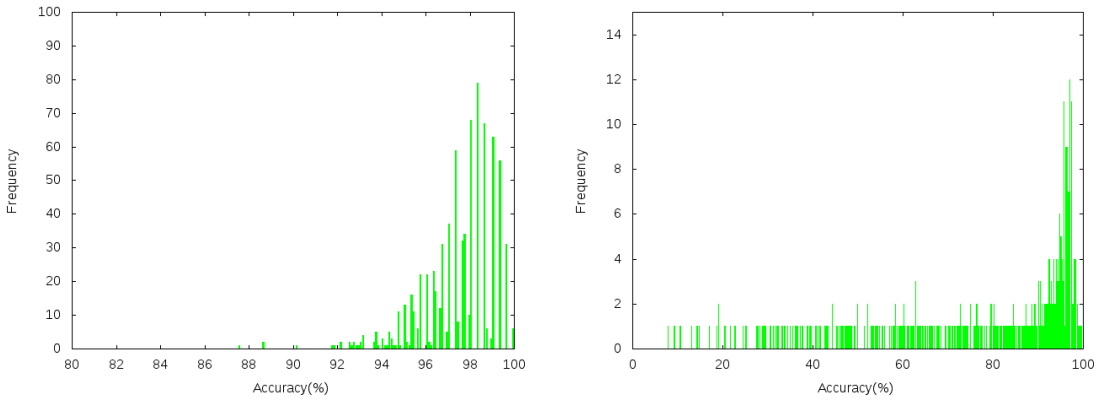
Processor	Without background processes	With background processes
i3	90.94	69.34
i5	94.7184	70.8012

6.2.1.1 *Multi-line probing*



(a) Without Background Processes (b) With Background Processes

Figure 6.11: Accuracy in i3 processor using multi-line probing in multi-core environment



(a) Without Background Processes (b) With Background Processes

Figure 6.12: Accuracy in i5 processor using multi-line probing in multi-core environment

Following table summarizes the accuracy obtained in case of probing multi-line in multi-core.

Table 6.6: Average accuracy obtained in i3 and i5 by probing multi-line in multi-core

Processor	Without background processes	With background processes
i3	99.2715	76.09
i5	97.4269	81.3817

The accuracy of probing multiple lines is more than that of probing single line as the victim and spy are executing parallel on different cores and as a result the probability of missing an operation is more if a single line is probed than in the case of probing multiple lines. Probing multiple lines is advantageous only when the victim can be interrupted multiple times during a single operation. Also the systems in which the experiments were performed, the clock frequency of the i3 processor is less than that of i5 processor and as a result the add and double operations take more time in i3 processor than i5 processor. So in i3 processor, the spy was able to interrupt the victim more number of times than i5 processor.

6.3 DSA

The attack on DSA in single core environment is done by Mr. Bholanth Roy so we performed experiments in multi-core environment.

6.3.1 Multiply and Square operations

As mentioned in 5.1 we need to obtain the square and multiply sequence and hence the time required for square and multiply operation on i5 processor was computed

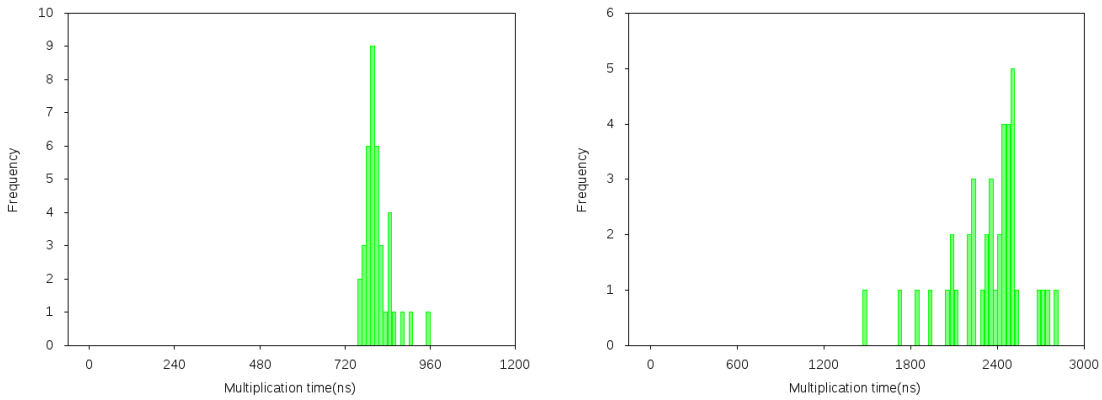
experimentally without spy and in presence of spy.The average time taken by these functions to execute is shown in below table:

Table 6.7: Average time(in ns) for square and multiply operation

	Square(in ns)	Multiply(in ns)
Without Spy	741	813
With Spy	2038	2332

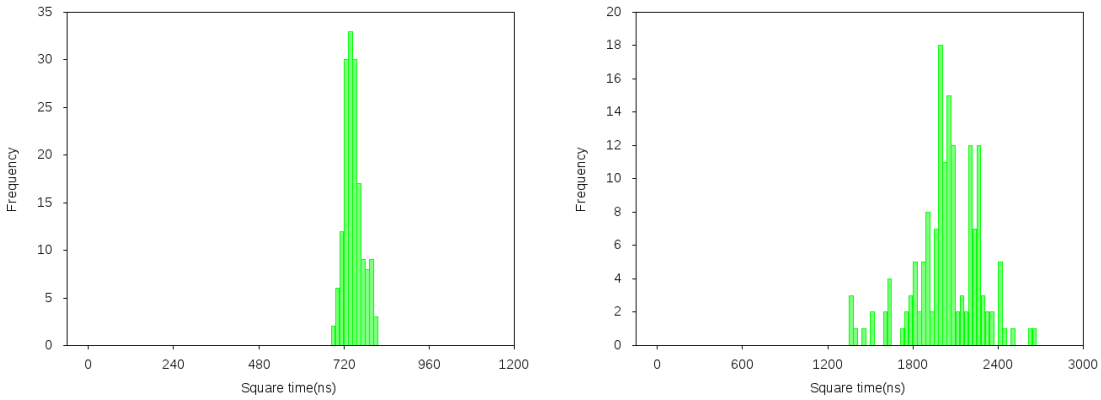
It can be observed from graphs 6.13a and 6.13b that the time required to compute multiply operation is much higher in presence of spy as compared to when only victim performs the multiply operation. This is because the spy threads flush the cache every-time they get scheduled. As a result the victim has to access the multiply function from the main memory resulting in increase in the access time.

Similarly, as shown below in graphs 6.14a and 6.14b, it can be observed that the time required to compute square operation is much higher in presence of spy as compared to when only victim performs the square operation. The reason for this is same as specified above.



(a) Without spy (b) With spy

Figure 6.13: Time required for multiply operation

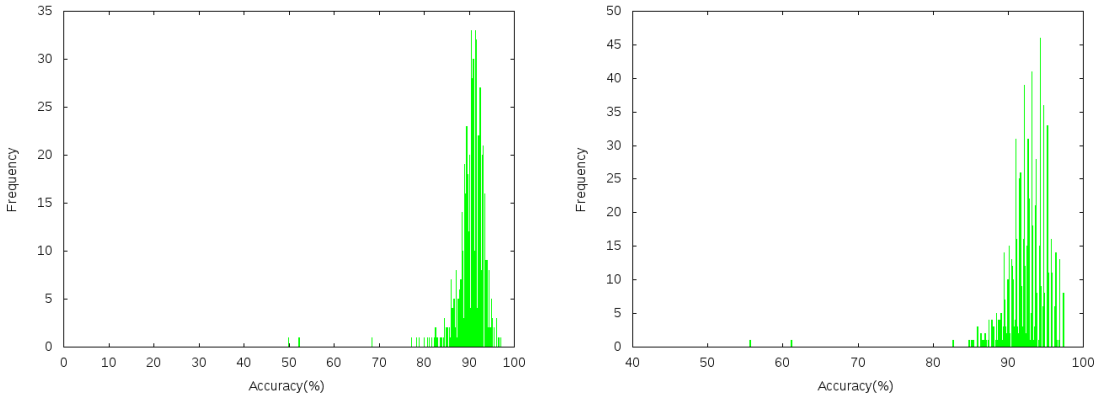


(a) Without spy (b) With spy

Figure 6.14: Time required for square operation

6.3.2 Attack on multiple cores

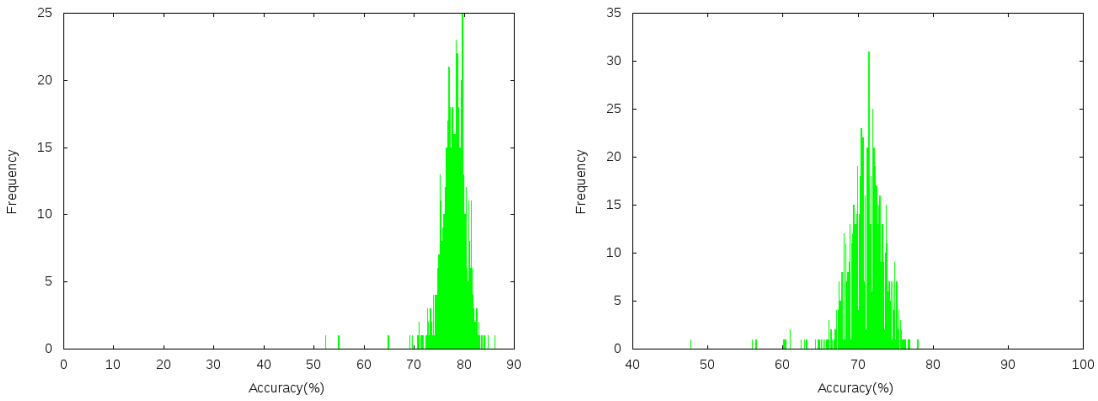
6.3.2.1 Single line probing



(a) i3 Processor (b) i5 Processor

Figure 6.15: Accuracy for DSA using single-line probing in multi core environment

6.3.2.2 Multi-line probing



(a) i3 Processor (b) i5 Processor

Figure 6.16: Accuracy for DSA using multi-line probing in multi core environment

Following table summarizes the accuracy obtained in multi-core architecture in DSA.

Table 6.8: Average accuracy obtained on i3 and i5 by in multi-core architecture in DSA

Processor	Single line probing	Multi-line probing
i3	90.34	77.9092
i5	92.4856	71.1495

As opposed to previous results, the accuracy is less while probing multiple lines than probing single lines as the spy was not able to interrupt the victim multiple times and the individual operations were not differentiated clearly. In ECDSA, this was not observed as the double and add operations take more time than square and multiply operation of DSA.

6.4 Observations

For ECDSA, the sequence obtained from side channel attack is then used to retrieve the private key using lattice attack[2]. For this, it is important to know the correctness of the sequence obtained. Through our experiments we obtained some results that can be helpful in determining this.
In a sample of 1000 ephemeral keys, we obtained the following results:

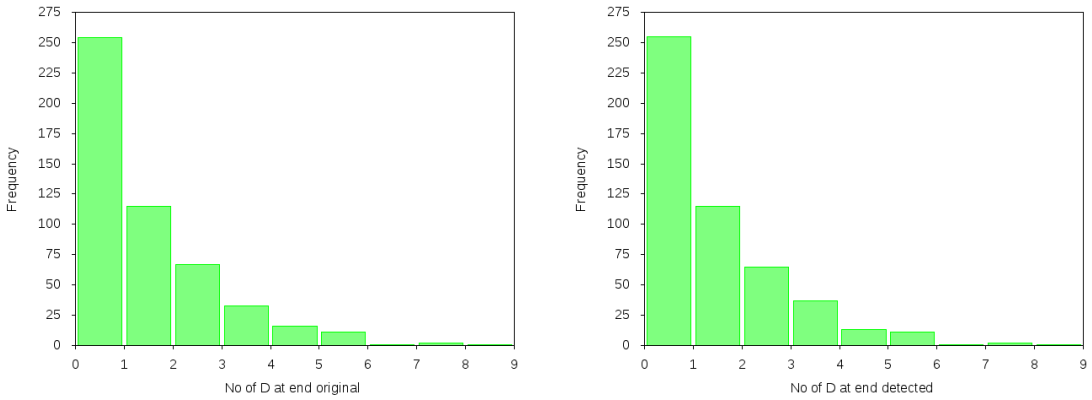
- The minimum length of the double-add sequence was observed to be 294 and the maximum length was observed to be 316.
- The minimum number of double operations in a single sequence were observed to be 245 and the maximum number of double operations were observed to be 257.

- The minimum number of add operations in a single sequence were observed to be 46 and the maximum number of add operations were observed to be 59.

Any sequence that do not fall into above categories can be considered to be wrong sequence and can be discarded for further experiments.

Also, OpenSSL uses window size of 4 for the scep256k1 curve used in ECDSA. As a result, in between any two A's if there are less than 4 D's in a sequence, that sequence can be considered as wrong sequence and can be discarded for further experiments.

In ECDSA, there is no one to one correspondence between the double and add sequence and the binary sequence of the ephemeral key as opposed to DSA. In DSA, the number of squares in the sequence directly gives the idea about number of zeros present in the binary. However this is not the case with DSA. But in ECDSA, the number of zeros after the last A directly translates to the number of zeros in the binary. This information can be used for lattice attack to obtain the private key. Following graph shows the number of Ds detected at the end in the original sequence and the detected sequence.



(a) Original Sequence (b) Detected Sequence

Figure 6.17: Ds at the end of the sequence

These graphs shows that the Ds detected at the end in original sequence and detected sequence in same in almost all the cases.

Chapter 7

Mitigation

This attack is implemented on the OpenSSL implementation of ECDSA for the curve secp256k1. During this attack a fixed private key-public key pair is used. Then we monitored via the FLUSH+RELOAD spy process the generation of a set of signature pairs(ri,si) for i=1,.....d. For each signature pair there is known hashed message value h and an unknown ephemeral key value ki. Using Flush+Reload side channel attack it is possible to get almost exact double and add sequence[2]. So in order to get full private key we need to feed this input lattice attack.This will require many set of k and sequence pair as input.

7.1 Limited use of a single private key

In order to get full private key this attack, combined with lattice require multiple signatures,so one way of mitigating it is limiting number of times a private key is used for signing. Bitcoin which uses the secp256k1 curve on which this attack focus, recommends using a new key for each transaction[4]. This recommendation however, is not always followed, exposing users to attack.

7.2 Speeding up scalar multiplication

Another option does not fully mitigate this attack but this method only reduce the effectiveness of the flush and reload attack. This is positive side effect of the speeding up scalar multiplication used in wNAF implementation of ECDSA.The GLV method is used to speed up the computation of the point scalar multiplication when elliptic curve has an efficiently computable endomorphism[6].

7.3 Using fixed window size

This attack is possible due to sliding window protocol in the wNAF algorithm for scalar multiplication. This can be prevented using a fixed window algorithm for scalar multiplication. This technique prevent any data flow from sensitive key data to memory access pattern sequence. There are other techniques that describes that there should not be any relationship between private key and the memory access sequence, NACL is also one of method to achieve this[3].

7.4 Disabling deduplication

Deduplication is a feature used to utilize the memory efficiently by sharing read-only pages. The side channel attack in multicore environment is possible as there is shared memory between the spy and the victim as in our case they use the same shared library OpenSSL. Disabling sharing of pages between spy and victim will mitigate the attack.

Chapter 8

Conclusion

In this work, partial information about ephemeral keys for ECDSA was obtained when spy and victim runs on the same core and also when they runs on different cores.The accuracy of this partial information i.e retrieval of double and add sequence was improved during the course of this work. Also, partial information about the ephemeral key was obtained when victim and spy was running on different cores in DSA.

The output of this work serves as an input to the various lattice techniques[2] to recover the private key. Due to increasing use of virtual machines, this work can be extended to perform side channel attack on Cross-VM platform[14] using the similar idea as used in multi-core side channel attack[7].

References

[1] Aciçmez, O., and Koç, Ç. K., 2006, “Trace-driven cache attacks on aes,”

[2] Benger, N., van de Pol, J., Smart, N. P., and Yarom, Y., 2014, “"ooh aah just a little bit" : A small amount of side channel can go a long way,” in *Cryptographic Hardware and Embedded Systems–CHES 2014* (Springer). pp. 75–92.

[3] Bernstein, D. J., Lange, T., and Schwabe, P., 2012, “The security impact of a new cryptographic library,” in *International Conference on Cryptology and Information Security in Latin America* (Springer). pp. 159–176.

[4] Bos, J. W., Halderman, J. A., Heninger, N., Moore, J., Naehrig, M., and Wustrow, E., 2013, “Elliptic curve cryptography in practice,” Cryptology ePrint Archive, Report 2013/734, `http://eprint.iacr.org/2013/734`.

[5] ECDSA, OpenSSL, `https://www.openssl.org`

[6] Gallant, R. P., Lambert, R. J., and Vanstone, S. A., 2001, “Faster point multiplication on elliptic curves with efficient endomorphisms,” in *Annual International Cryptology Conference* (Springer). pp. 190–200.

[7] Irazoqui, G., Eisenbarth, T., and Sunar, B., 2015, “S $ a: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes,” in *Security and Privacy (SP), 2015 IEEE Symposium on* (IEEE). pp. 591–604.

[8] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B., 2014, “Wait a minute! a fast, cross-vm attack on aes,” in *International Workshop on Recent Advances in Intrusion Detection* (Springer). pp. 299–319.

[9] Kerrisk, M., 2010, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, 1st ed. (No Starch Press, San Francisco, CA, USA). ISBN 1593272200, 9781593272203

[10] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B., 2015, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy*, pp. 605–622.

[11] Neve, M., and Seifert, J.-P., 2006, "Advances on access-driven cache attacks on aes," in *International Workshop on Selected Areas in Cryptography* (Springer). pp. 147–162.

[12] Pereida García, C., Brumley, B. B., and Yarom, Y., 2016, ""make sure dsa signing exponentiations really are constant-time"," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16 (ACM, New York, NY, USA). pp. 1639–1650.

[13] Pthreads, POSIX Threads Programming, `https://computing.llnl.gov/tutorials/pthreads/`

[14] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S., 2009, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09 (ACM, New York, NY, USA). pp. 199–212.

[15] Roy, B., Giri, R. P., Ashokkumar, C., and Menezes, B., 2015, "Design and implementation of an espionage network for cache-based side channel attacks on aes," in *Proceedings of the 12th International Conference on Security and Cryptography*, pp. 441–447.

[16] Yarom, Y., and Benger, N., 2014, "Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack.." *IACR Cryptology ePrint Archive* **2014**, 140.

[17] Yarom, Y., and Falkner, K., 2014, "Flush+ reload: a high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 719–732.