Review and implement cache timing attacks

Mini Project
Noam Hever
Nadav Zada

TABLE OF CONTENTS

Github for this project:
https://github.com/Noam32/Cache-timing-attack-project
Drive folder with project's files
https://drive.google.com/drive/folders/17TfK_bw8LbwpgIauJcGYzqGZokCH-Mjx?usp=sharing

Background and presenting the concept

Definitions:

Covert channel - a hidden or 'illegal' communication medium that was not designed to be used for communication or is possibly breaking the system's security rules and policies (i.e. Separating between processes).

Side Channel - The use of an established covert channel by an attacker to deceive a victim program/process and either cause it to malfunction or to discover secret information about its state and inner variables.

Examples for such channel:

- Timing (time of execution)
- Power (which we used in assignments 4 and 5)
- Thermal emanations
- EM emanation(radiation) (lecture 9 and 12)
- Acoustic emanation

We will focus on the Timing channel - our goal is to break the logical separation of the memory protection system, and transfer information between processes, in a way that isn't permitted by the system security policy.

The channel can be used in both directions - the victim might send information to the attacker. Or the attacker might send information to the victim which will change its runtime execution (state or behavior).

The advantage of Timing attacks is that it doesn't require measurement equipment and/or physical access to the device. Can be done solely in software (and possibly a network connection).

Six possible directions for timing side channel attacks:

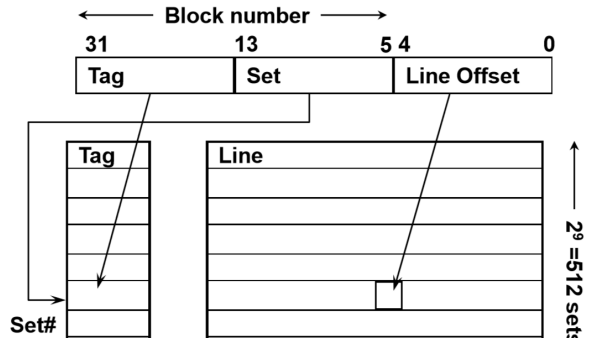
1. I**nstruction with Different Execution Timing** - we saw this in lecture 8 - (double and add / multiplication and squaring)
   [An easy solution - make sure that the program finishes the execution in a constant time - independent of instructions used]
2. **Variable Instruction Timing**- some instructions take a different time to execute depending on the state of the CPU.
   [loads/stores, Rng register accesses]
3. **Functional Unit Contention** - processes may use a hardware/software resource and it will be unavailable to other processes - and that can leak information about its internal execution.
4. **Stateful Functional Units** - after a process uses a hardware unit - it might leave that unit in a state which - if observed by another process might leak information about the action/calculation made.
5. **Prediction Units** - we learned about them in Computer architecture - they make predictions according to the history of last executed instructions and runtime state. Prediction units might be shared between processes - and the attacker might be able to view the branching history of the victim. Alternatively, its branching decisions might affect the branch predictions for the victim.
6. **Memory Hierarchy** - If the CPU has a cache - there is a variable read/load time depending on whether the value is stored in cache or not - and where they are in the memory hierarchy (L1,L2,L3,Ram,Disk).

We will focus on **Memory Hierarchy - cache based timing channel attacks.**

The attacks we are trying to implement in simulation are:
Prime+Probe, Flush+Reload, Evict+Time, or Cache Collision Attack.

Short introduction about “**cache sets” (Eviction sets)**:

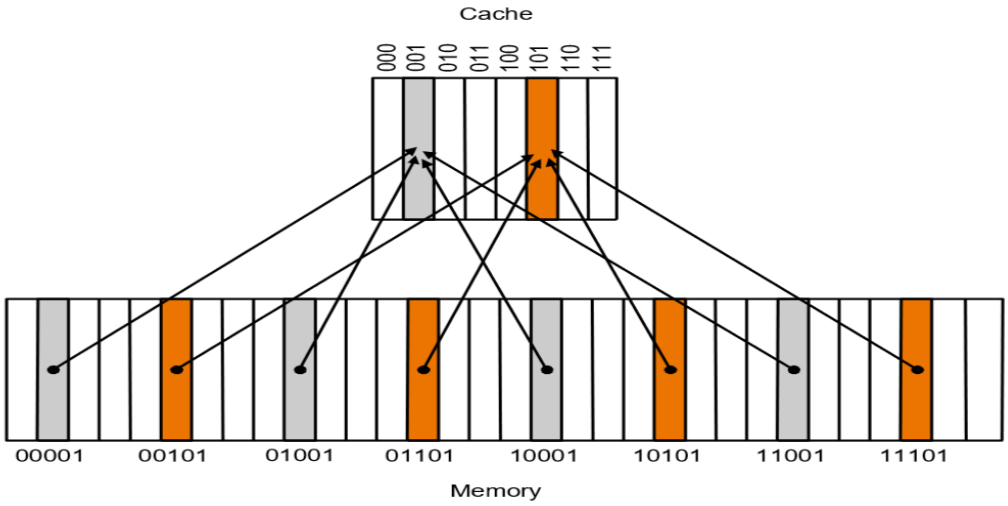
When mapping a block from memory to a line in cache - we look at the “set” which is a part of the address, for example:



Each line in cache is mapped to multiple blocks from memory that have the same “set”.

The blocks which are mapped to the same line are considered to be in the same “Eviction Set”.

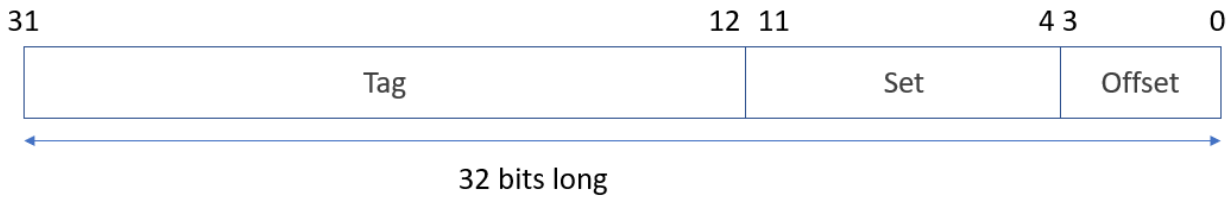
For example: each gray block in memory will be mapped to set “001” in cache. (All of the gray sets are in the same Eviction set).



Many cache attacks depend on the ability of the attacker to find these eviction sets efficiently.

If we want to store a block in the cache, in its corresponding line, we will have to evict another block (which is from the same Eviction set) from that line in the cache. (There are caches with multiple ways and various eviction policies).

In our simulation we will use a cache that has one way (direct map) - and is 4KB in size (size of data array).



Prime and Probe

(video :Prime & Probe Cache Attack - YouTube and GitHub - Yossioren/AttacksonImplementationsCourseBook - course from Ben Gurion university)

“Prime” = putting the cache in a known state. We can do this by filling up a particular set in the cache (might have more than one block if we have n-way associative cache) by accessing addresses in our memory space - that correspond to that set.

(Note: It’s possible to know the mapping by reverse engineering - for our simulations we will assume that the attacker has acquired such knowledge from publicly available material or by reverse engineering by himself).

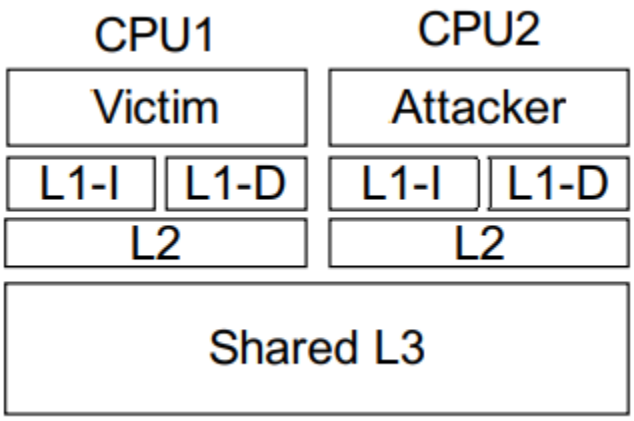
Then, we “probe” - meaning that we access these addresses and measure the time it took us to access them. We can tell whether or not they are in the cache, because then the read time is significantly shorter than Dram access (not in cache).
We can wait for the victim to start running (or call its execution when possible) and then when the wait time is over - try to read the addresses again and if the read is now significantly slower - we know that the victim had used an address from the set we occupied.
We can repeat this for all the sets in cache until we know all of the sets which the victim is using.

Flush and reload attack

sources:

https://www.youtube.com/watch?v=UmLB1EWelCw

https://eprint.iacr.org/2013/448.pdf



This attack is focused on the LLC - last level cache. In this attack we can even attack processes that run simultaneously on 2 separate cores - since the last level cache is shared between them. The idea is that the two processes - victim and attacker are sharing memory pages. This could be either due to previously mentioned reverse engineering or because both processes are using a library (i.e Rsa encryption methods) and both processes are pointing to the file/page that has the execution code for that library.

The attacker will want to use that to learn about the execution of the victim process by evicting data/instruction lines and then observing the read times for that shared line.

The attack has 3 phases: [for each cache lines]

1. Flush - flushing the monitored line (using commands assembly like “clflush”).
2. Waiting - and letting the victim process to run.
3. Reloading - trying to read the line we evicted and measuring how long the read took.

We expect the read to take a long time - because we evicted the line. But, If during the waiting time- the victim has used/accessed that cache line - then that line will be in cache and our read will be fast. (We have to use the same line the victim is using - to not evict it).

This attack is better than the ‘prime and probe’ attack because it allows us to tell if a specific memory line was accessed by the victim.

Previously in ‘prime and probe’ we could only tell if the victim had accessed a memory location/block from the cache set (which could be millions of bytes).

We can use the knowledge of what lines were reached during the victim's runtime - to get information about the results of the calculation. For example - by evicting lines which are connected to instructions in the victim’s code that are conditionally executed (depending on branch resulousion) we can tell if the line was executed or not -and therefore know information about inner values of the calculation.

An example from the paper and video: [Exponentiation by Square-and-Multiply - used in RSA] if we evict the lines that have instructions 8,9 - and then reloading - we can determine whether or not in that loop iteration e_i=1 or e_i=0 - and therefore know information about the value of ‘x’ which is secret. (The loop is iterating over the bits “e_i” -s of e - usually part e could be the public key or the secret key since we use the exponent function for encrypting and decrypting).

```
1  function exponent(b, e, m)
2  begin
3    x ← 1
4    for i ← |e| − 1 downto 0 do
5      x ← x^2
6      x ← x mod m
7      if (e_i = 1) then
8        x ← xb
9        x ← x mod m
10     endif
11   done
12   return x
13 end
```

The generated encryption system consists of:

- The public key is the pair (n, e).
- The private key is the triple (p, q, d).
- The encrypting function is $E(m) = m^e \bmod n$.
- The decrypting function is $D(c) = c^d \bmod n$.

If we discover the bits of the exponent ‘d’ while calculating C^d - we can recover the value of ‘d’ - and discover the private key which is secret.

In our simulation - we would want to attack the shared location (cache lines) in which the lines 8 and 9 are stored. We can flush these lines - and we would be able to tell if the lines were accessed.

If the reload takes a short time - then we know what line was accessed - and we can invert the sbox (we know the input to the sbox - or at least the range of values for the input).

Evict+Time

https://www.researchgate.net/publication/262389651_Security_testing_of_a_secure_cache_design

Unlike Flush **+** Reload**,** this attack doesn't require any known shared data or memory pages.
Also, we do not need the flush "clflush" instruction to be available (non-privileged).
Similarly, to prime and probe - we evict a specific set from the cache.
Then we wait for the victim process which executes symmetric encryption to run (if possible - by calling it, if not - waiting for it to get cpu time).
And then we measure the runtime of the victim process. If the line, we evicted was used during the runtime of the process - then we expect to have a cache miss - which will result in a longer runtime for the victim process.

We will need to measure the time before we start 'messing' with the cache to achieve a baseline of what the runtime should be (We can do it multiple times and average the results).

This is slightly different from prime +probe - in which the attacker measures the time it takes himself - to read back the values which he has brought previously to the cache. Here we are measuring the total runtime of the victim process and trying to understand if that runtime is longer if a specific line was evicted.
Like prime and probe - We are only learning the Eviction sets in which the sensitive data is stored - not what the data is or where exactly it is stored (Eviction Set: groups of virtual addresses that map to the same cache set -these could possibly be millions of addresses).

Cache Collision attack

(https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.4753&rep=rep1&type=pdf)
(https://www.comparitech.com/blog/information-security/what-is-a-collision-attack/#What_is_a_collision_attack)

A collision attack is simply when an attacker finds some kind of collision and uses it to undermine the security that the cryptographic scheme was assumed to provide.

There are Many different types of examples of collision attacks, a lot of them deal with schemes that involve using hash functions (MD5, SHA-1, SHA-2,SHA-3). The approach in this attack is to utilize the correlation between cache hits and encryption time. It is a model for attacking AES by using the timing effects of cache-collisions to gather noisy information about the likelihood of relations between key bytes (in our simulation we will focus on the first round of AES).

The principle is that the runtime will vary depending on whether we had “cache collisions” - meaning that sbox lookups are accessing the same line - which means we got a cache hit.

We have a Cache-Collision Assumption - for any pair of lookups i, j, given a large number of random AES encryptions with the same key, the average time when ⟨li⟩ = ⟨lj⟩ (upper bits of sbox’s input) will be less than the average time when ⟨li⟩ ≠ ⟨lj⟩. It means that if the victim accessed the same set-in cache several times, the attacker could realize it by seeing the little access time, and conclude that this byte keeps the secret information of the encryption.

“These attacks are different from timing attacks because they require that the attacker gain direct knowledge about cache access patterns - thus they are directly using cache accesses as a cryptographic side-channel instead of timing.”

We are attacking the first round of AES encryption - where the [k_i $\oplus p0$] is the input to the lookup table.

The goal is to learn information about the secret key’s bytes. In our simulation we implemented the “first round attack” - which gives you information about the xor results between key bytes that are at indexes from the same “family “.
(i.e key bytes at indexes that are plugged into the same Sbox table - for example the bytes K[0], K[4],K[8],K[12] are xored with plain [0,4,8,12] and the result is used to access the T0 sbox table. So K[0], K[4],K[8],K[12] are in the same “family).

We will learn the values of K[0] xor K[4] , K[0] xor K[8], K[0] xor K[12],K[4] xor K[8],K[4] xor K[12], K[8] xor K[12] . We don’t get the key itself -but this information reduces the number of guesses for recovering the key by half.

This is the most complex attack we implemented - we added more explanations in the implementation chapter. (page 29).

Simulation

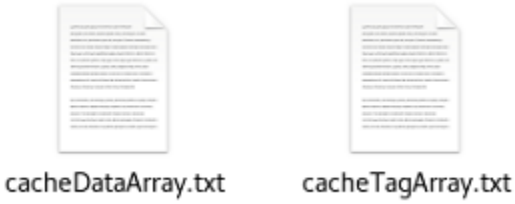
Firstly, let's present the simulation set and how we built it.

The simulation was developed and ran on a machine running CentOS Linux provided by the university.

The code was written in C in visual studio code and compiled using gcc.

The cache is simulated. We are using a text file as the virtual cache.

Actually, the cache is separated into 2 files: One for the tag array, and one for the data array.



Each time a process is running - given control over the CPU - we run a method that reads from the files - and creates an array representing the current state of the cache.

As the process is running it is accessing the simulated cache (and reading /writing/flushing /evicting it) and changing its state.

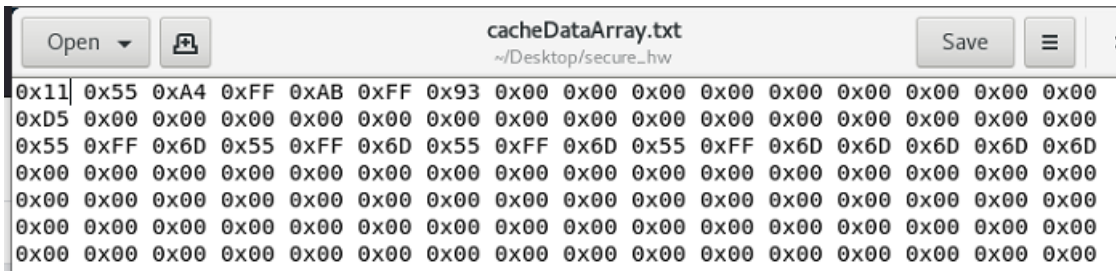
Once the process “leaves” the CPU - either when it finishes execution or if it is waiting it will write the array back into the text files so that the other process will have the updated version of the cache.

(For example - the attacker waits in flush and reload attack for the victim to run)

The cache looks (visually) like this:

set (hex)	Tag(hex)	Data Array
0x00	0x00005	0x10 0x98 0x76 0x15 0x98 0x76 0x56 0x14 0x18 0x28 0x71 0x15 0x58 0x16 0x56 0x09
0x01	0x000F0	0x10 0x98 0x76 0x15 0x98 0x76 0x56 0x14 0x18 0x28 0x71 0x15 0x58 0x16 0x5S 0x09
0x02	0x02400	0x10 0x98 0x76 0x15 0x98 0x76 0x56 0x14 0x18 0x28 0x71 0x15 0x58 0x16 0x56 0x09
0x03	0x60110	0x10 0x98 0x76 0x15 0x98 0x76 0x56 0x14 0x18 0x28 0x71 0x15 0x58 0x16 0x56 0x09
0x04	0x00100	0x10 0x98 0x76 0x15 0x98 0x76 0x56 0x14 0x18 0x28 0x71 0x15 0x58 0x16 0x56 0x09
0x05	0x00800	0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

And this is how the files look:



Open cacheTagArray.txt ~/Desktop/secure_hw

```
0x54ADC
0x54ADC
0xFFFFF
0x00000
0x00000
0x00000
0x00000
```

Cache size and how the address is split:

We have decided to use in most attacks the following cache (Direct map):

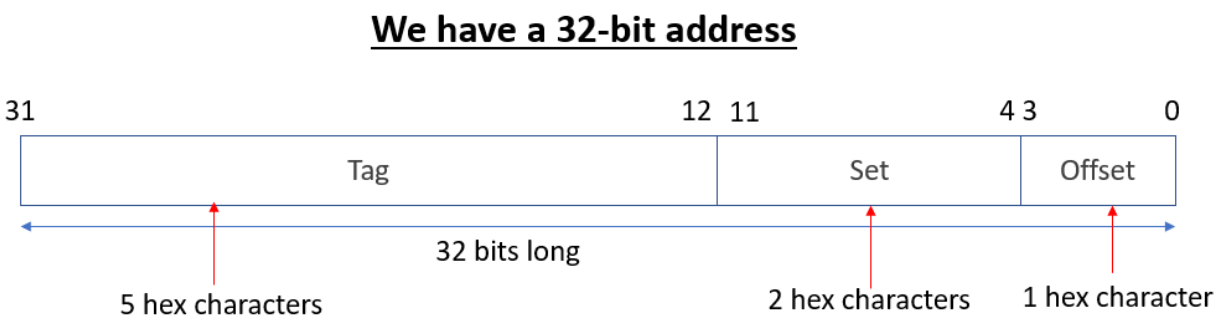
Data array size:

block size = 16 bytes => offset is 4 bits.

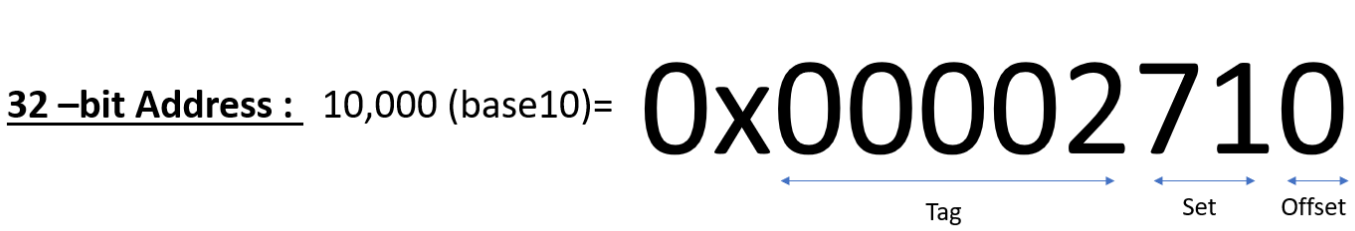
Number of lines in cache = 256 => Set is 8 bits.

=> Therefore, Tag is 20 bits (32 - 8- 4)

So, this will be how the addresses are split:



For example:



The cache is 4Kb in size (size of the data array).

Additionally each process will have its own “virtual global memory “- an array of 64Kb .

```
//global arrays for the programmer level
unsigned char * globalMem ;
int pointerTounAllocatedMemory=0;
```

```
#define global_mem_size 65536
```

```
//defining the global memory space (in DRAM) for this process:
globalMem=(unsigned char *)malloc(global_mem_size*(sizeof(unsigned char)));
```

At the start of each process - the data is loaded into the memory (without reading it -this is just a setup) The process will read and write to the memory - and the state of the cache will change accordingly (hit/ miss/ evicting flushing etc).

Example:

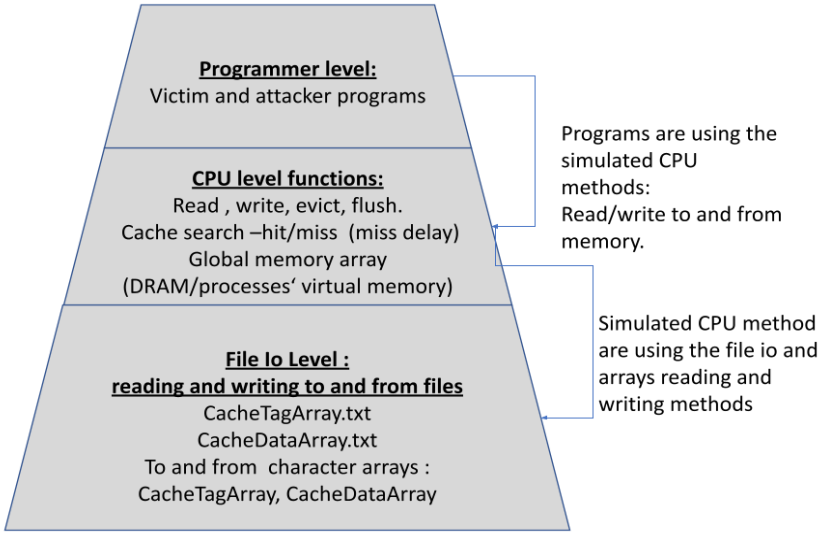
```
unsigned char key=memRead((int)&globalMem[offsetOfKeyInGlobalMem]);
```

Note: our simulation doesn’t have virtual and physical addresses and they are all “physical”.

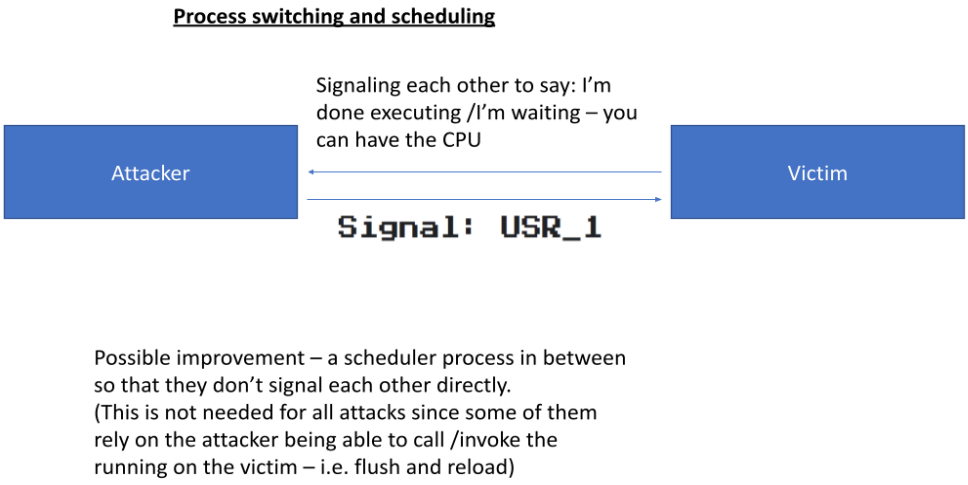
Levels of abstraction

As we explained, each of the victim and the attacker will be a process. When a process has control over the CPU, it can do operations that involve memory executions. The processes will use CPU methods, and they are executed by operations on files and arrays, as we can see in this diagram:

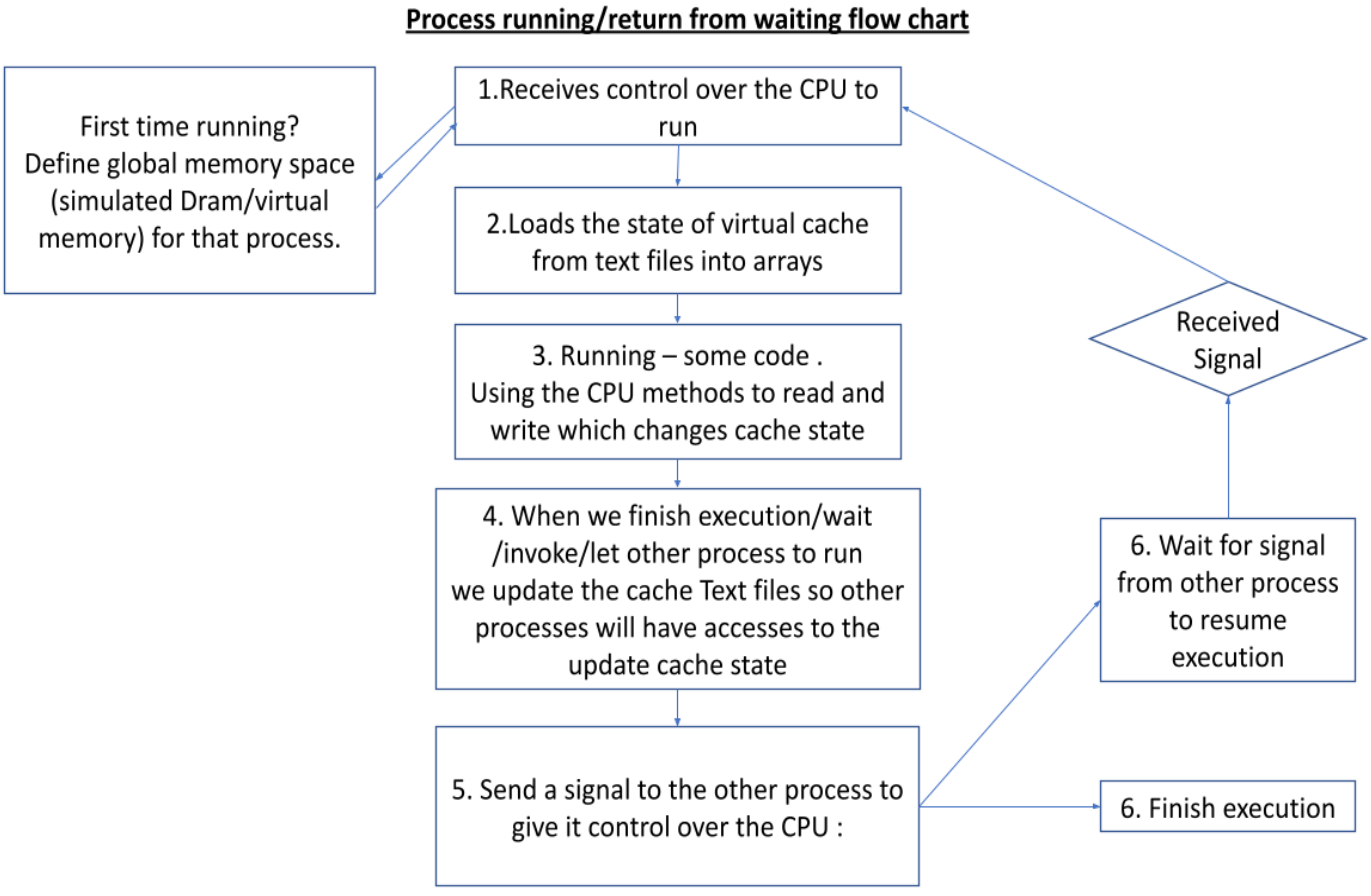
3 levels of abstraction :



The communication between the victim and the attacker (processes) will be with signals, as described in this diagram:



The flow for process running or returning from waiting is described in this diagram:



Functions and API for the simulation's levels:

Full function list is in the appendix at page 40 full source code starts at page 46

Organization of the project's files:

CODE FILES FOR OUR PROJECT

Attackers and victims for each attack

Attackers	Attack type	Victims
attacker_prime.c	Prime and probe	Victim_prime.c
attacker_flush.c	Flush and reload	Victim_flush.c
attacker_evict_time.c	Evict and time	Victim_evict_time.c
attacker_collision.c	Cache collision	Victim_collision.c

File IO level functions and CPU level functions .
Used by all attacker and victim files and included in them

fileIOFunctions.h

Text files that store information and process communication

Cache

cacheDataArray.txt

cacheTagArray.txt

Communicating process PID

Victim_PID.txt

Attacker_PID.txt

Sending plaintext to encrypt (cache collision)

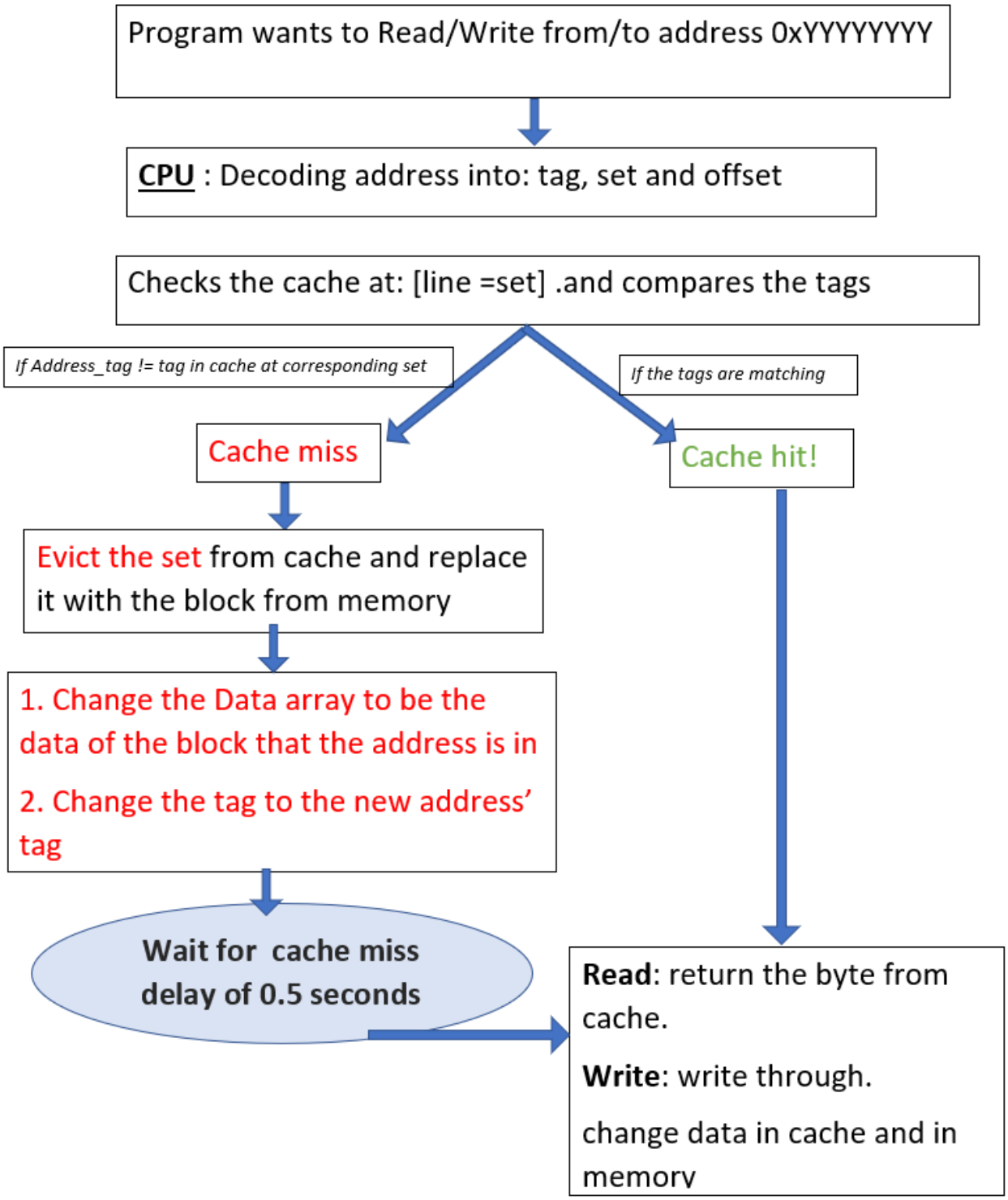
plainTextToEncrpt.txt

Contents of a shared memory page (flush and reload)

SharedMemoryPage.txt

Memory read/write flow:

When the program (victim and attacker) reads or writes to memory - we go through the following flow (Identical flow to the ones shown in the pdf's and taught in Computer architecture course):



Example:

If we read from 17 consecutive addresses (bytes) - we should have a miss at the first read, which will be followed by 15 hits and then another miss (assuming the first address is aligned to the start of a block -> %16 =0).

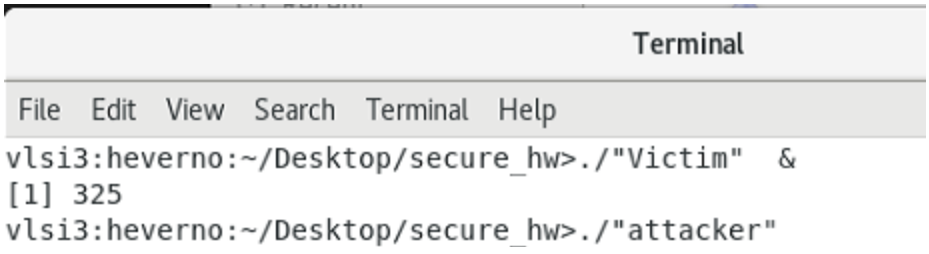
Since the miss penalty time was decided by us to be 0.5 seconds - the first and last reads take about 0.5 seconds to complete, and the other reads take a very short time (around 0.0001s sec).

See example below: (reading starts at address 12972080 - which is aligned)

```
readByteAtAddress: cache miss:evicting set 3 ,new tag is :0x00C5F
Total time = 0.500108 seconds
readByteAtAddress:CACHE HIT ON ADDRESS 12972081 SET 3 and Tag =0x00C5F
Total time = 0.000003 seconds
readByteAtAddress:CACHE HIT ON ADDRESS 12972082 SET 3 and Tag =0x00C5F
Total time = 0.000002 seconds
readByteAtAddress:CACHE HIT ON ADDRESS 12972083 SET 3 and Tag =0x00C5F
Total time = 0.000002 seconds
readByteAtAddress:CACHE HIT ON ADDRESS 12972084 SET 3 and Tag =0x00C5F
Total time = 0.000002 seconds
readByteAtAddress:CACHE HIT ON ADDRESS 12972085 SET 3 and Tag =0x00C5F
Total time = 0.000002 seconds
readByteAtAddress:CACHE HIT ON ADDRESS 12972086 SET 3 and Tag =0x00C5F
Total time = 0.000001 seconds
readByteAtAddress:CACHE HIT ON ADDRESS 12972087 SET 3 and Tag =0x00C5F
Total time = 0.000002 seconds
readByteAtAddress:CACHE HIT ON ADDRESS 12972088 SET 3 and Tag =0x00C5F
Total time = 0.000003 seconds
readByteAtAddress:CACHE HIT ON ADDRESS 12972089 SET 3 and Tag =0x00C5F
Total time = 0.000001 seconds
readByteAtAddress:CACHE HIT ON ADDRESS 12972090 SET 3 and Tag =0x00C5F
Total time = 0.000002 seconds
readByteAtAddress:CACHE HIT ON ADDRESS 12972091 SET 3 and Tag =0x00C5F
Total time = 0.000002 seconds
readByteAtAddress:CACHE HIT ON ADDRESS 12972092 SET 3 and Tag =0x00C5F
Total time = 0.000002 seconds
readByteAtAddress:CACHE HIT ON ADDRESS 12972093 SET 3 and Tag =0x00C5F
Total time = 0.000002 seconds
readByteAtAddress:CACHE HIT ON ADDRESS 12972094 SET 3 and Tag =0x00C5F
Total time = 0.000002 seconds
readByteAtAddress:CACHE HIT ON ADDRESS 12972095 SET 3 and Tag =0x00C5F
Total time = 0.000003 seconds
readByteAtAddress: cache miss:evicting set 4 ,new tag is :0x00C5F
Total time = 0.500092 seconds
```

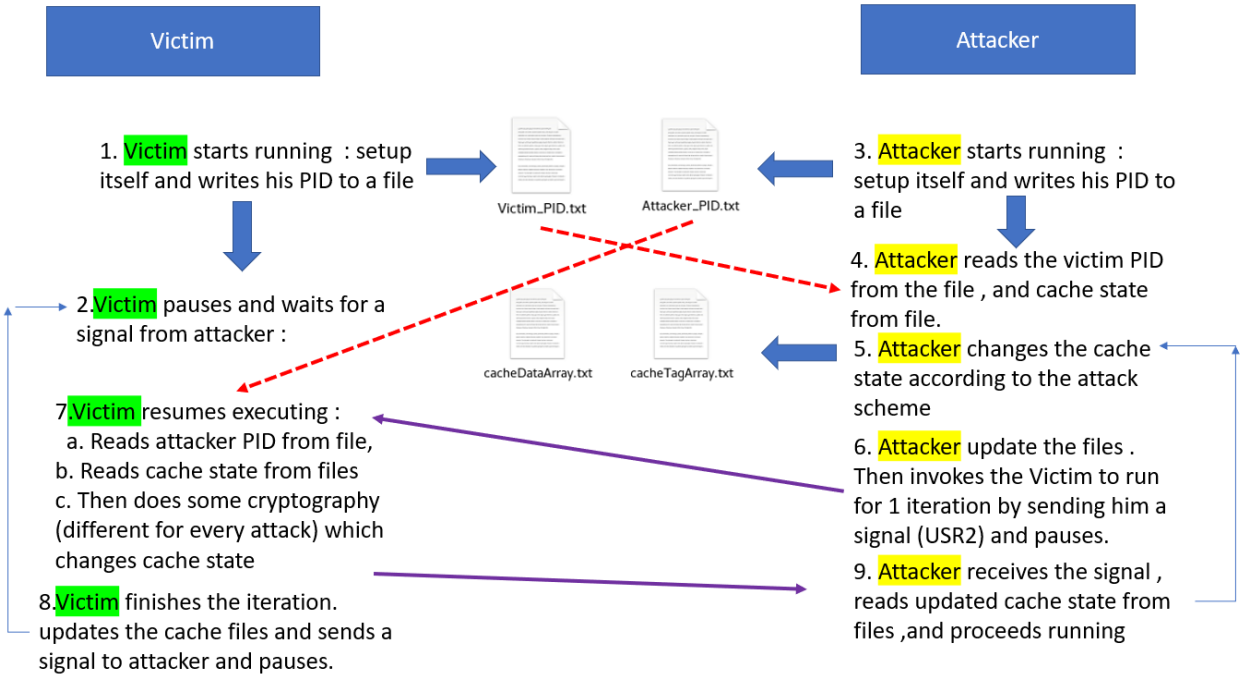
Explaining the process switching by signaling:

Firstly, both files Victim.c and attacker.c are compiled. Then we ran them in this order from the terminal.



...

The processes are communicating and switching control over the virtual CPU and cache by files and signals.This flowchart shows the communication process:



The processes are getting each other's PIDs from the files - and whenever they decide to switch control and let the other process run they:

1. Update the cache files (cache was changed during their runtime and we want the other process to get the updated cache state).
2. Send a signal to the other process (We get the PID from the file).
3. Pause and wait to get a signal back to resume executing.
4. When signal arrives - read updated cache state from files -and resume running.

FIRST ATTACK:

Prime and Probe:

The running process of the attack consists of 4 steps:

(1) Prime - in the attacker process we evict all of the sets - one by one. After doing that, the cache sets are in a “known state”.

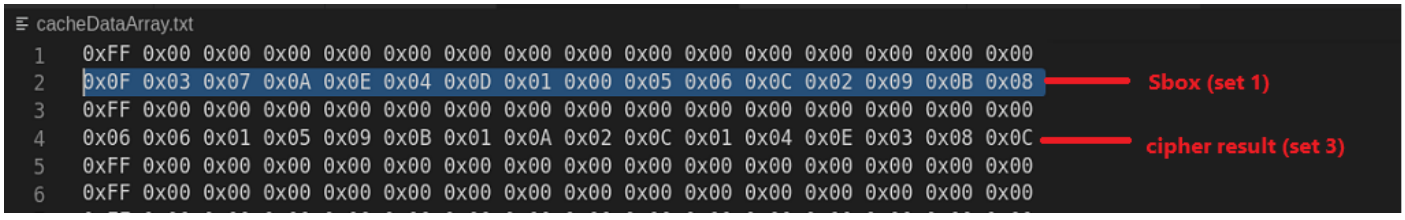
(2) Measure access time before letting the victim run. This step is necessary for the Comparison of times that will give us the information we want to learn - which sets in cache the victim is using. We want to be able to tell whether a set was changed by the victim.

(3) Let the victim run - the victim is doing some cryptographic action - xor between a plaintext and secret key. Then the victim is calculating the ciphertext by outputting the sbox’s result in the xor result index. The secret key, sbox and cipher are located in the cache.

(4) measure access time after - after the victim runs, the control over the CPU returns to the attacker process, and it measures the access time to the same set he primed before letting the victim to run.

If the measured time is very close to the one we measured in step 2 it means that the victim did not use that set in cache. On the other hand, if we got significantly larger time it means that the victim did use that set in cache, and therefore there was a cache miss that caused the memory access time to increase.

In these figures we can see the data cache after the victim has done the cryptographic action, and stored the secret key, sbox and cipher in the cache.



(Note 1: the row numbers in screenshots are always +1 from the set index because text editors start counting from 1).

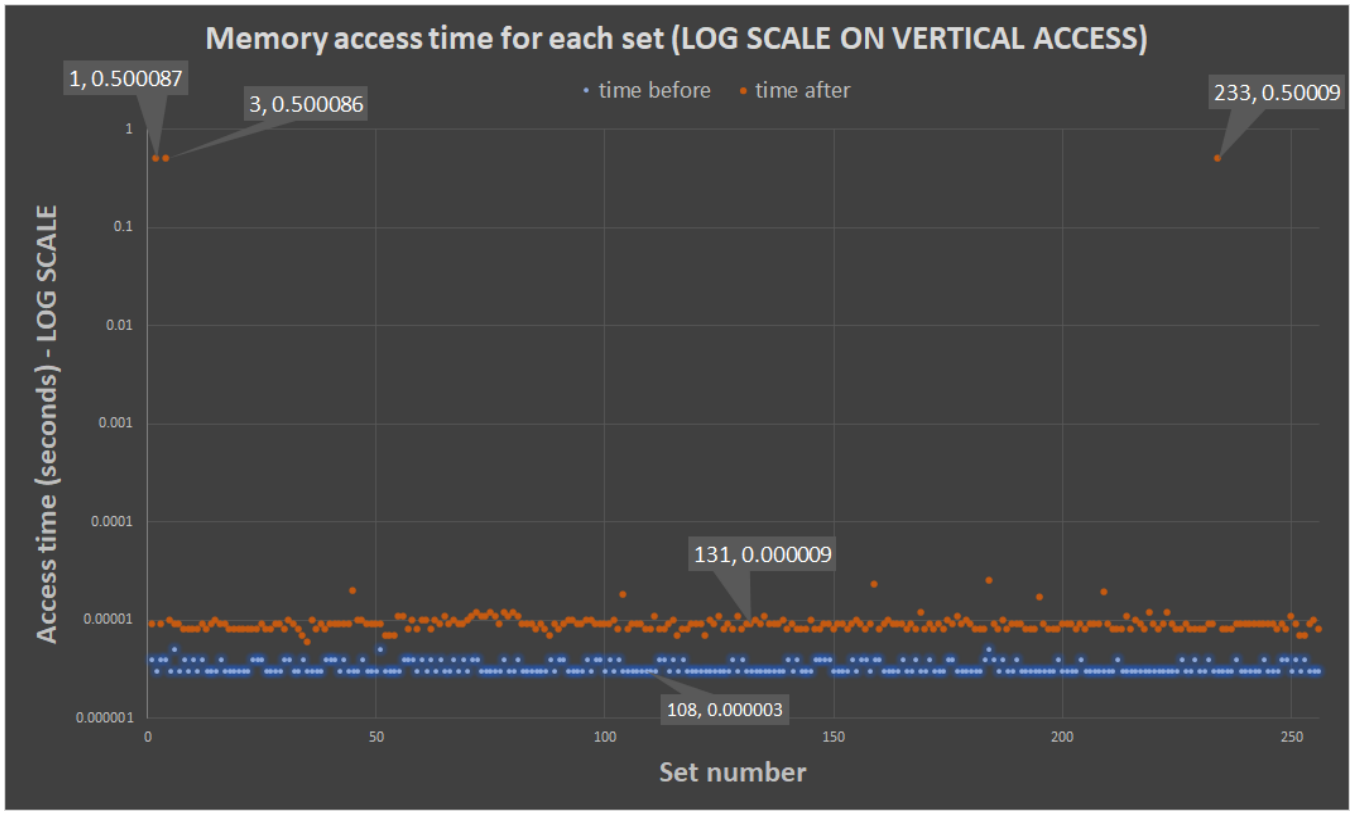
(Note 2: the plaintext is randomized and isn't shown in cache, Sbox and key are chosen by us and hard coded in global memory).

Results:

We printed the times and copied the output from the terminal to a file. Then we used excel to plot the graph. The vertical axis is presented in log scale to better show the difference in orders of magnitude.

We can see that the 3 sets which the Victim was using: 1, 3, 233 (access time is longer - about 0.5 seconds - which is the miss penalty we decided on).

We (the attackers) do not know what is in these sets (is it the key /sbox or cipher or some inner variables that are not important to us).



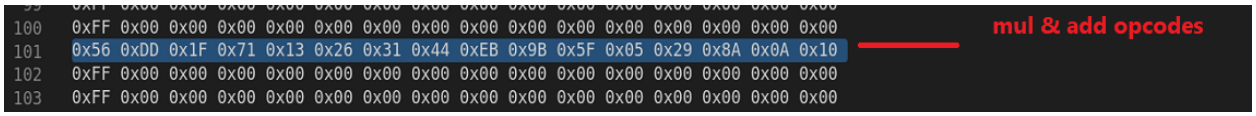
Either way, as explained in the theoretical part, we know the "Eviction sets" which the information is stored at (We know that the Victim is using a memory address with set "01" - the problem is that there are millions of addresses with the corresponding set - that is the weak point of this attack). (As explained - it is using 3 blocks - one for the sbox, one for the cipher, and one for the secret key).

Flush and Reload

What needed to add a feature to our simulation in order to implement this attack:

We added a read only shared memory page (Which in our limited simulation is only 16 bytes). This memory page is accessible to both victim and attacker. It is simulating the shared memory page of the RSA encryption libraries that is mentioned in the paper and video from the theoretical part). We are using a different method to access the shared memory to ensure that both processes have the same fixed address (0x00002640) for the start of the page (also, the contents of the shared memory are stored in a text file - and both processes have access to it).

The simulated instructions: the opcodes are place holders - the shared memory will be accessed each time we execute a specific/sensitive part of the code - but we didn't implement instruction decoding. The main idea is that if multiply and add lines are executed then this line will be brought to cache by the victim - so we can simulate "flush and reload".



ReadFromSharedMemPage() : (Instead of the usual memRead() / memWrite())

```
/the shared page only has the instructionsOpcodes for  MultplyAndAdd
/offset is range 0-15 (inclusive).
nsigned char ReadFromSharedMemPage(int offsetInSharedPage,char **cacheTagArray,char **cacheDataArray){
   int simulated_Location_Of_page=0x00002640; //maps to set 100 . tag =ABCDE .offset 0 (aligned)
   unsigned char instructionsOpcodes[16] ={0x56,0xDD,0x1F,0x71,0x13,0x26,0x31,0x44,0xEB,0x9B,0x5F,0x05,0x29,0x8A,0x0A,0x10};
```

What is the victim doing: it is running RSA decryption with exponent function for one byte - The processes are switching control after each iteration of the for loop - and the attacker then "reloads" the memory and checks the time it took (short time -> cache hit -> e_i =1 for the previous iteration).

Decrypting cipher with exponent function using private key

Cipher d_private key N = modulo value

```
1  function exponent(b, e, m)
2  begin
3    x ← 1
4    for i ← |e| − 1 downto 0 do
5      x ← x^2
6      x ← x mod m
7      if (e_i = 1) then
8        x ← xb
9        x ← x mod m
10     endif
11   done
12   return x
13 end
```

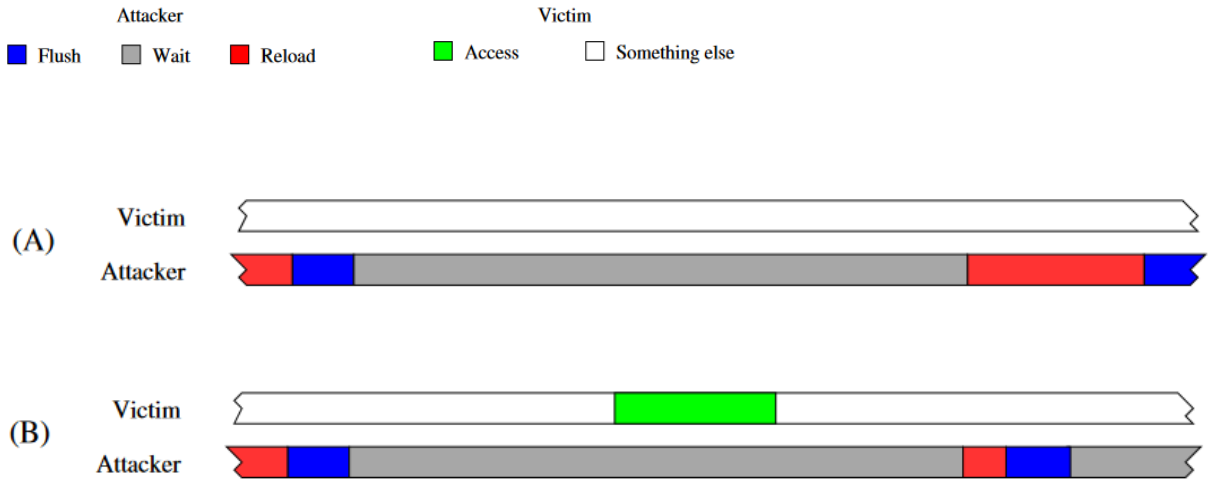
Result = c^d mod n =
(plain^e)^d mod n =
plain ^(e*d) mod n =
plain^1=plain.

What will the attacker do (for 32 iterations):

(1) flush the cache line that has the critical code lines (set 100 in our simulation).

(2) Then it will let the Victim run for one iteration.

(3) When the victim gives back the control -> the attacker will reload the address of the critical code lines and measure the time it took to perform the read (Short time = cache hit - the Victim has executed the critical lines).



[Note: in the Academic paper (page 4) there were 2 more scenarios - we didn't implement them because they require parallel execution which our simulation isn't capable of simulating].

We Implemented the exponent(b,e,m) function from the paper:

```
//exponent(b,(d=e),m ):
for( i=31;i>=0;i--){ //exponent is 32 bits long
    x=pow((double)x,(double)2);
    x=x%m;
    int d_i=getSpecificBitFromInteger(private_SecretKey_d,i);
    if(d_i==1){
        //We excecute the if->we are accessing the line in shared memory -
        //and the block will be read and loaded into cache
        ReadFromSharedMemPage(0,cacheTagArray,cacheDataArray);
        x=x*b;
        x=x%m;
    }
```

```
1  function exponent(b, e, m)
2  begin
3    x ← 1
4    for i ← |e| − 1 downto 0 do
5      x ← x^2
6      x ← x mod m
7      if (e_i = 1) then
8        x ← xb
9        x ← x mod m
10     endif
11   done
12   return x
13 end
```

attacker:

```
//we flush all of the sets -one by one - and then we invoke the victim to run .
//then we try to read the data again - and measure the time it took to read.
//(1) flush  (2) wait for other procees to run (3) Reloading and measuring time
int sharedAddress=0x00002640;
int setToflush =getAddress_Set(sharedAddress); //The attacker knows the shared address and it's set.
for(i=31;i>=0;i--){
    flush(setToflush,cacheTagArray,cacheDataArray); //((1)
    //now we let the victim to run:(2)
    UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
    signalOtherProccessToRun(victimPid);
    pause();//wating for victim to return the control to us.
    //Getting the updated cache state:
    cacheTagArray = loadCacheTagFromFile();
    cacheDataArray=loadCacheDataArrayFromFile();
    //(3) Reloading
    timeToAccess_criticalLines[i]=measureTimeToReadFromAddress(sharedAddress);
}
```

Results:

We chose the private key for the victim:

```
int private_SecretKey_d=0xFFABCD91;// (exponent e) - 32 bit
```

We ran the attack and printed all of the bit guesses based on time (if access took more than 0.5 seconds, the miss penalty we decided on, then we had a cache miss. Otherwise -cache hit).

```
int didWeHaveAshortAccessTime=timeToAccess_criticalLines[i]<0.5;
if(didWeHaveAshortAccessTime)  //short time ->cache hit ->the victim accessed the critical lines ->d_i was 1.
    guessedBits[i]=1;
else                                //long time ->cache miss ->victim didnt accesses critical lines-> d_i was 0.
    guessedBits[i]=0;
```

```
****************The results are**********************
Access time on iteration 1 is 0.000016 guessed bit =1
Access time on iteration 2 is 0.000010 guessed bit =1
Access time on iteration 3 is 0.000009 guessed bit =1
Access time on iteration 4 is 0.000008 guessed bit =1
Access time on iteration 5 is 0.000009 guessed bit =1
Access time on iteration 6 is 0.000009 guessed bit =1
Access time on iteration 7 is 0.000010 guessed bit =1
Access time on iteration 8 is 0.000008 guessed bit =1
Access time on iteration 9 is 0.000010 guessed bit =1
Access time on iteration 10 is 0.500111 guessed bit =0
Access time on iteration 11 is 0.000009 guessed bit =1
Access time on iteration 12 is 0.500084 guessed bit =0
Access time on iteration 13 is 0.000010 guessed bit =1
Access time on iteration 14 is 0.500085 guessed bit =0
Access time on iteration 15 is 0.000011 guessed bit =1
Access time on iteration 16 is 0.000010 guessed bit =1
Access time on iteration 17 is 0.000012 guessed bit =1
Access time on iteration 18 is 0.000010 guessed bit =1
Access time on iteration 19 is 0.500084 guessed bit =0
Access time on iteration 20 is 0.500084 guessed bit =0
Access time on iteration 21 is 0.000009 guessed bit =1
Access time on iteration 22 is 0.000010 guessed bit =1
Access time on iteration 23 is 0.500094 guessed bit =0
Access time on iteration 24 is 0.000012 guessed bit =1
Access time on iteration 25 is 0.000009 guessed bit =1
Access time on iteration 26 is 0.500090 guessed bit =0
Access time on iteration 27 is 0.500084 guessed bit =0
Access time on iteration 28 is 0.000010 guessed bit =1
Access time on iteration 29 is 0.500084 guessed bit =0
Access time on iteration 30 is 0.500085 guessed bit =0
Access time on iteration 31 is 0.500081 guessed bit =0
Access time on iteration 32 is 0.000009 guessed bit =1
guessedKeyInBinary: 11111111101010111100110110010001
RecoveredKey in decimal -5517935 . in hex :FFABCD91
*****************Attacker MAIN ENDED SUCCESSFULLY !**********************
vlsi3:heverno:~/Desktop/secure_hw>
```

As you can see - we were able to recover the full 32-bit secret key using flush and reload attack on a known shared address. (In green - binary representation. In yellow - hex representation).

Evict and time

We start by running and measuring the runtime of the victim process 100 times.

We need to measure the time before we start ‘messing’ with the cache to achieve a baseline (average) of what the runtime should be.

```
i= 90 . measureOtherProcessRuntime : 0.031707
i= 91 . measureOtherProcessRuntime : 0.020474
i= 92 . measureOtherProcessRuntime : 0.022549
i= 93 . measureOtherProcessRuntime : 0.019911
i= 94 . measureOtherProcessRuntime : 0.039586
i= 95 . measureOtherProcessRuntime : 0.021751
i= 96 . measureOtherProcessRuntime : 0.021206
i= 97 . measureOtherProcessRuntime : 0.039519
i= 98 . measureOtherProcessRuntime : 0.020746
i= 99 . measureOtherProcessRuntime : 0.021684
Average runtime basline is : 0.040073
```

Since miss delays are relatively long compared to the runtime (0.5 seconds) we needed to use more measurements to reduce the effect of the few misses we get on the first runs.

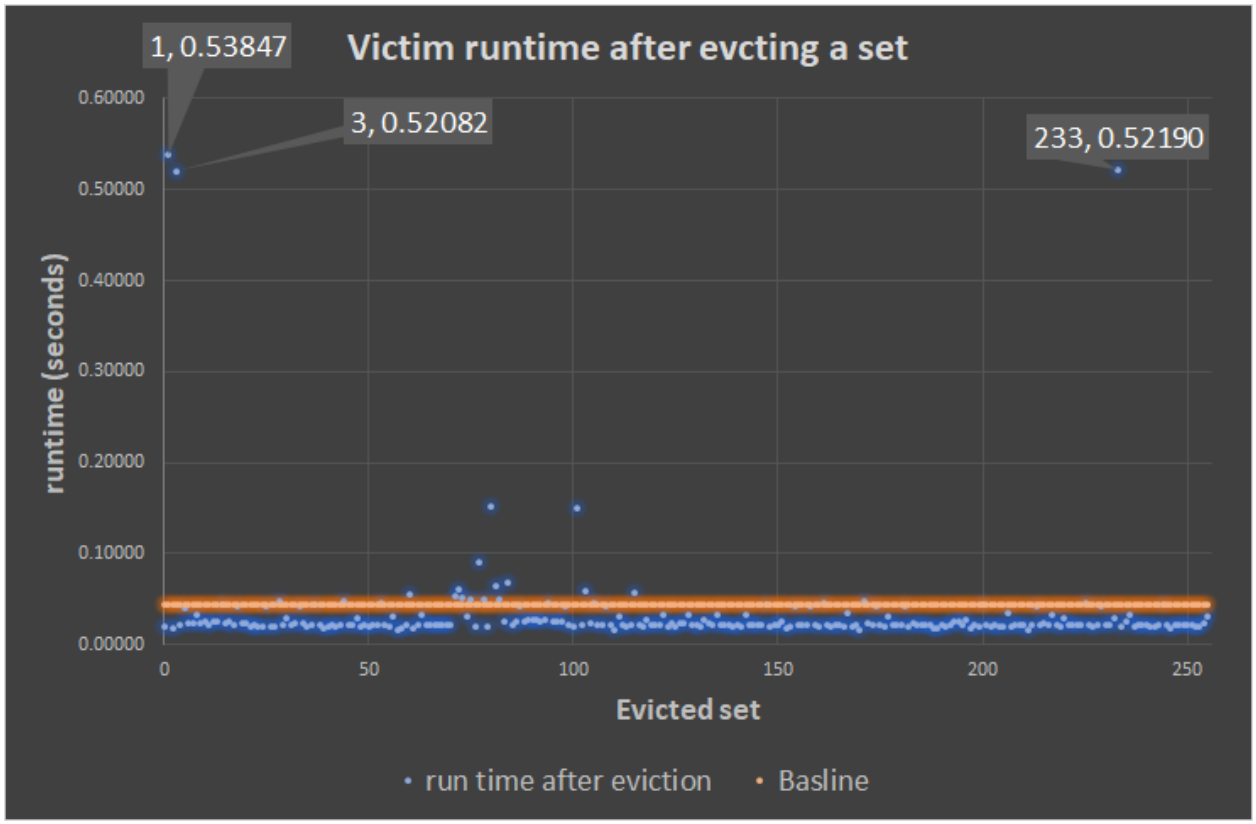
The victim is executing the same DoSomeCrypotography() that was used in prime and probe: (Calculating Sbox (key XOR plain [i]) for a 16-byte random plain) each time it is invoked by the attacker.

Now we evict each set (0-255) and let the victim run and measure its runtime.

If the victim runtime is longer than 0.5 seconds - we will understand that there was a cache miss in the victim that was caused by the eviction of the set - and the attacker will conclude that the set is used by the victim.

Result:

Note (as in prime and probe -the victim uses 3 sets, set **1** -sbox, **3**-cipher ,**233** - secretKey).



We can see that the attacker recognized the 3 sets that the Victim was using.

A note about system noise for process runtime timing:

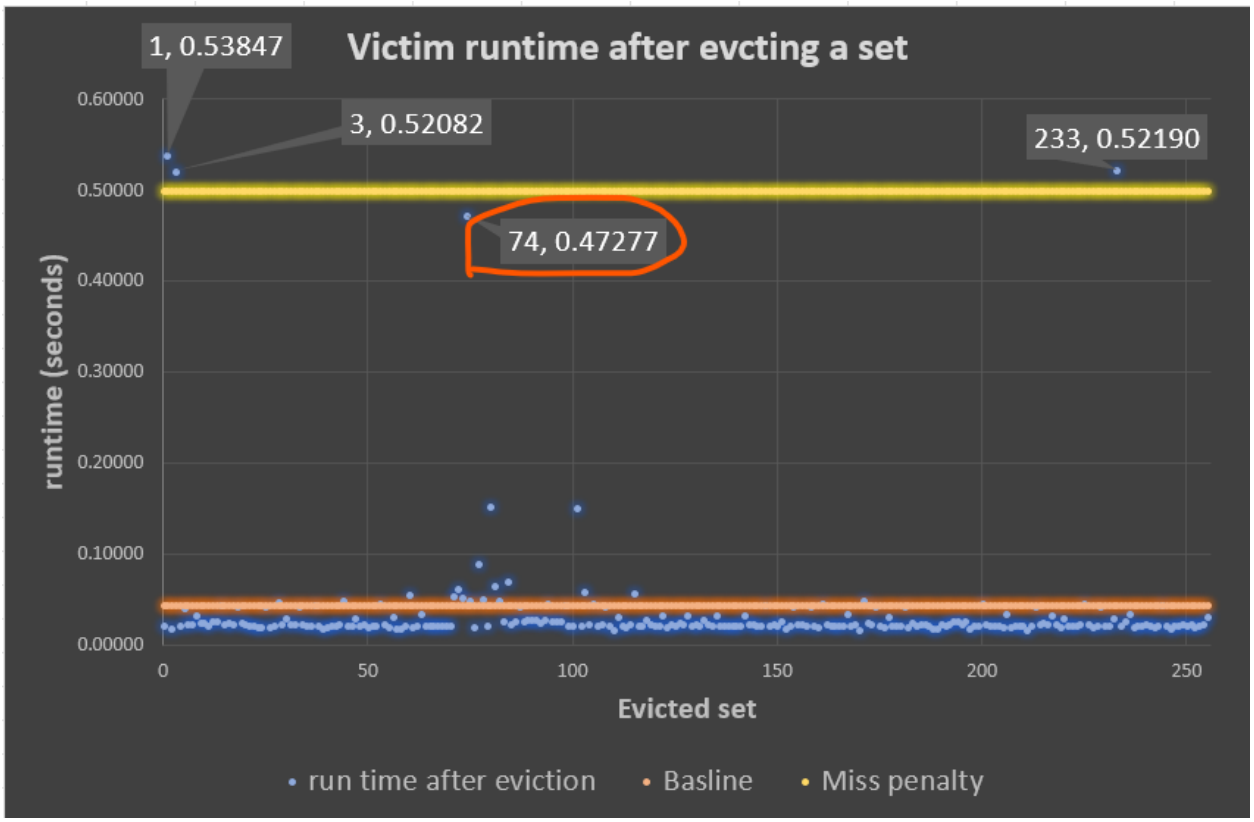
During our test we have encountered an anomaly: a specific set that we assumed should have a short runtime (74) - had a runtime of 0.47 seconds (almost as long as the miss penalty time - which will make the attacker guess that it is a set that the victim is using!).

After checking the logs and making sure that indeed, there was no cache miss on that set during the runtime of the victim, we ran the test a few more times, and got normal results - short runtime of about 0.02 seconds.

We have concluded that the error was caused by “system noise “- meaning that process switching in a linux server we were using causes a delay when returning to the attacker. (Something happened with the scheduler - might have been due to user activity during runtime - which caused the system to let other processes run and delay the execution of our two processes).

Solution: measure time and evict for each set multiple times - to reduce the effect of a single noisy measurement (maybe use statistical noise filtering).

Example for a noisy result on one measurement: (yellow -miss penalty, orange -baseline)



<u>To better demonstrate the effects of the attack we want to use more than just 3 sets:</u>

When we are using more sets - we expect to get more cache misses during the victim runtime.

We would use an extended version of **DoSomeCryptography()** that uses a 16 byte key that is <u>stored in 16 memory blocks</u> (one byte in each block - the rest of the block is uninitialized).

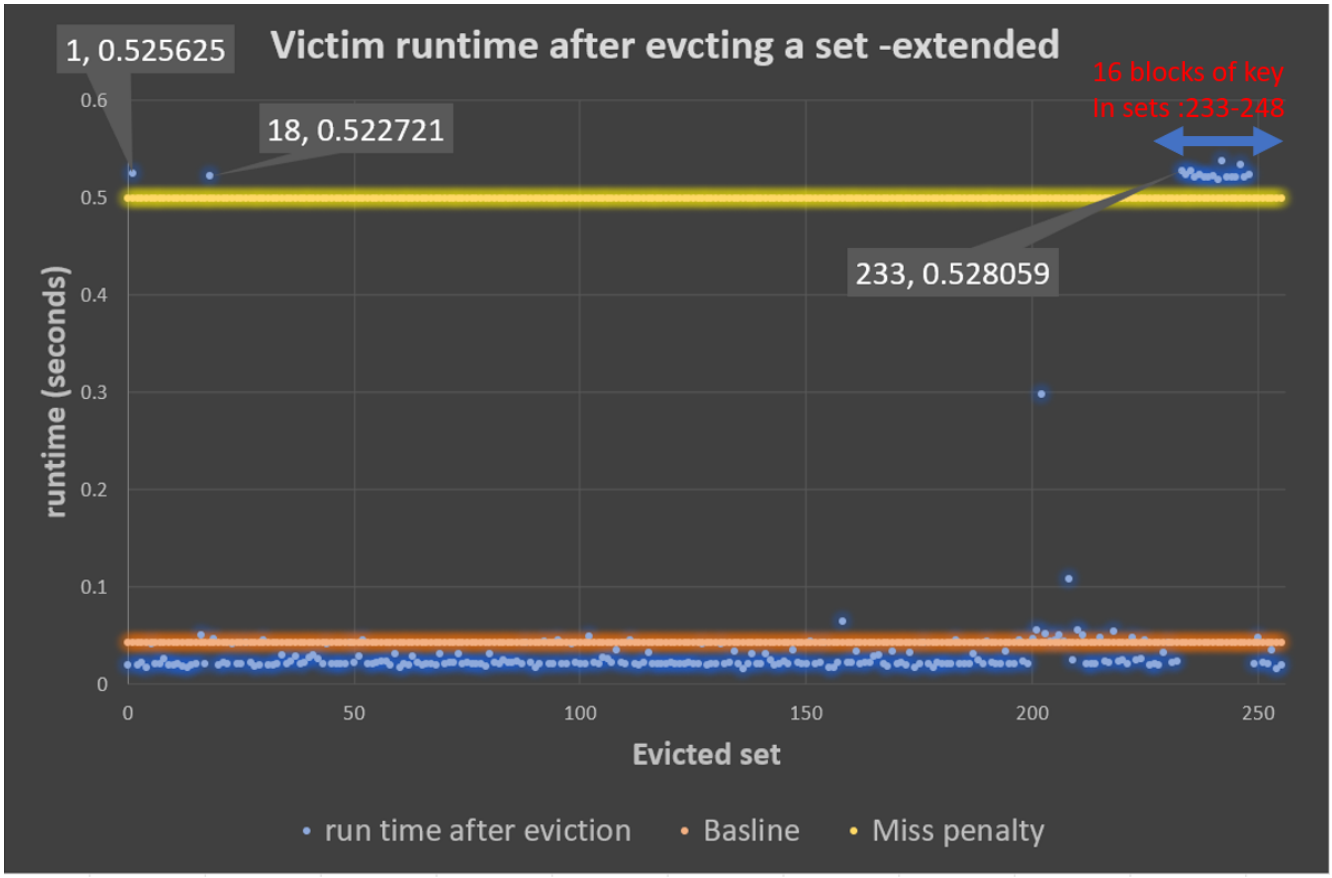
The sets the victim is using now are (1 - sbox, 233-248 Keys, 18 - cipher):

The key can be seen here in the cacheDataArray.txt (index are +1 because the editor starts at 1).

```
234   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xEA 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
235   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x55 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
236   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x43 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
237   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xAF 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
238   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x68 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
239   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x91 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
240   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xDB 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
241   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xCC 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
242   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xE5 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
243   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x48 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
244   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x03 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
245   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x26 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
246   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x31 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
247   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x69 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
248   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x88 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
249   0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xBB 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

Results:

(Note: the cipher was moved from set 3 to 18 because of the memory allocation routine in our simulation).



Cache collision attack

We will simulate the first part of the attack explained in:

https://link.springer.com/content/pdf/10.1007/11894063_16.pdf

Under "**First Round Attack**".

In this attack the attacker has access to an encryption oracle - which is the victim process.

The attacker selects plaintexts which it chooses in order to test for differences in encryption runtime (the runtime of the victim) and sends them to the victim to be encrypted.

The principle is that the runtime will vary depending on whether we had "cache collisions" - which means that two sbox lookups are accessing the same line - which causes a cache hit.

Since in the first round of the AES encryption the xor between plaintext byte and key byte goes straight into an Sbox - we will want to attack the first round.

X0_i = pi $\oplus$ k -> X is plugged into the Sbox as an indice -> memory access to: Sbox(X_i).

We will attack the T0 Sbox. (One of the s-boxes in AES) which will be stored in memory.

```
unsigned char Sbox [256]=
                {   0x63 ,0x7C ,0x77 ,0x7B ,0xF2 ,0x6B ,0x6F ,0xC5 ,0x30 ,0x01 ,0x67 ,0x2B ,0xFE ,0xD7 ,0xAB ,0x76 ,
                    0x10 ,0xCA ,0x82 ,0xC9 ,0x7D ,0xFA ,0x59 ,0x47 ,0xF0 ,0xAD ,0xD4 ,0xA2 ,0xAF ,0x9C ,0xA4 ,0x72 ,
                    0xC0 ,0x20 ,0xB7 ,0xFD ,0x93 ,0x26 ,0x36 ,0x3F ,0xF7 ,0xCC ,0x34 ,0xA5 ,0xE5 ,0xF1 ,0x71 ,0xD8 ,
                    0x31 ,0x15 ,0x30 ,0x04 ,0xC7 ,0x23 ,0xC3 ,0x18 ,0x96 ,0x05 ,0x9A ,0x07 ,0x12 ,0x80 ,0xE2 ,0xEB ,
                    0x27 ,0xB2 ,0x75 ,0x40 ,0x09 ,0x83 ,0x2C ,0x1A ,0x1B ,0x6E ,0x5A ,0xA0 ,0x52 ,0x3B ,0xD6 ,0xB3 ,
                    0x29 ,0xE3 ,0x2F ,0x84 ,0x50 ,0x53 ,0xD1 ,0x00 ,0xED ,0x20 ,0xFC ,0xB1 ,0x5B ,0x6A ,0xCB ,0xBE ,
                    0x39 ,0x4A ,0x4C ,0x58 ,0xCF ,0x60 ,0xD0 ,0xEF ,0xAA ,0xFB ,0x43 ,0x4D ,0x33 ,0x85 ,0x45 ,0xF9 ,
                    0x02 ,0x7F ,0x50 ,0x3C ,0x9F ,0xA8 ,0x70 ,0x51 ,0xA3 ,0x40 ,0x8F ,0x92 ,0x9D ,0x38 ,0xF5 ,0xBC ,
                    0xB6 ,0xDA ,0x21 ,0x10 ,0xFF ,0xF3 ,0xD2 ,0x80 ,0xCD ,0x0C ,0x13 ,0xEC ,0x5F ,0x97 ,0x44 ,0x17 ,
                    0xC4 ,0xA7 ,0x7E ,0x3D ,0x64 ,0x5D ,0x19 ,0x73 ,0x90 ,0x60 ,0x81 ,0x4F ,0xDC ,0x22 ,0x2A ,0x90 ,
                    0x88 ,0x46 ,0xEE ,0xB8 ,0x14 ,0xDE ,0x5E ,0x0B ,0xDB ,0xA0 ,0xE0 ,0x32 ,0x3A ,0x0A ,0x49 ,0x06 ,
                    0x24 ,0x5C ,0xC2 ,0xD3 ,0xAC ,0x62 ,0x91 ,0x95 ,0xE4 ,0x79 ,0xB0 ,0xE7 ,0xC8 ,0x37 ,0x6D ,0x8D ,
                    0xD5 ,0x4E ,0xA9 ,0x6C ,0x56 ,0xF4 ,0xEA ,0x65 ,0x7A ,0xAE ,0x08 ,0xC0 ,0xBA ,0x78 ,0x25 ,0x2E ,
                    0x1C ,0xA6 ,0xB4 ,0xC6 ,0xE8 ,0xDD ,0x74 ,0x1F ,0x4B ,0xBD ,0x8B ,0x8A ,0xD0 ,0x70 ,0x3E ,0xB5 ,
                    0x66 ,0x48 ,0x03 ,0xF6 ,0x0E ,0x61 ,0x35 ,0x57 ,0xB9 ,0x86 ,0xC1 ,0x1D ,0x9E ,0xE0 ,0xE1 ,0xF8 ,
                    0x98 ,0x11 ,0x69 ,0xD9 ,0x8E ,0x94 ,0x9B ,0x1E ,0x87 ,0xE9 ,0xCE ,0x55 ,0x28 ,0xDF ,0xF0 ,0x8C
                };
```

We would try all $\langle p_i \rangle \oplus \langle p_j \rangle = \langle k_i \rangle \oplus \langle k_j \rangle$ xor result options to try to discover the xor results between 4 bytes from the secret key.

The indices (X-s) that go into the T0 table are:

$$< P_i > \oplus < K_i >$$

$$i \in \{0,4,8,12\}$$

By trying all 16 combinations for the plaintext's bytes xor results we will be able to discover: <K_0>⊕ <K_4>, <K_0>⊕ <K_8>, <K_0>⊕ <K_12>, <K_0>⊕ <K_4>, <K_4>⊕ <K_8>, <K_8>⊕ <K_12>.

A chart might better explain this:

GOAL:

We want to calculate :

$$< K_0 > \oplus < K_4 > , < K_0 > \oplus < K_8 > , < K_0 > \oplus < K_{12} > , \ < K_4 > \oplus < K_8 > , < K_4 > \oplus < K_{12} > , < K_8 > \oplus < K_{12} > ,$$

Call Victim to perform an encryption for Plaintext that is constant except for two bytes : P_i and P_j .

(Plain is 16 bytes long).

We will repeat this for all 16 possible values of , $< P_i > \oplus < P_j >$ - 16 options .($< P_{j,i} > are\ 4\ bits\ long$) .

(fix upper , $< P_j >$ and set $< P_i >$ to the value we want) .

We will measure the runtime of the victim for each option –and save it a matrix -> Time (i, j , x) (i,j,x∈{0…15})

if we get a significantly shorter runtime:

it means that the value we chose for , $< P_i > \oplus < P_j >$ is equal to , $< K_i > \oplus < K_j >$.

If we repeat this for every i , j – we can learn the values of all pairs of many key's bytes pair. (i , j)

This attack does not discover the key itself - only reduces the complexity and the number of keys guesses an attacker needs to perform (compared to brute force). Since knowing the result of a bitwise xor between two bits reduces the options for their values from 4 to 2.
(if A xor B = 1, We conclude that A! =B. Therefore, the options for A=b=1 and A=b=0 are not possible, and we can only have A=1, B=0, or A=0, B=1.
In other words: If A xor B =1 -> A! =B. If A xor B = 0 -> A=B).

The size of the <Ki>s, <Xi>s and <Pi>s is the size of byte (8 bits - used as an indices to map to 256 byte sbox) is determined by the size of the offset in memory/cache block (which isn't reflected in a cache collision and therefore we don't discover any information about it. Since access to the same block will always result in a cache hit).
In our simulation we have 16-byte blocks - therefore the offset is 4 bits long and therefore <Ki>s, <Xi>s and <Pi>s are 4 bits long.
This means that if we discover: [<Ki> xor < Kj >] - we are only discovering the four MSBs of the xor result between Ki and Kj.

In order to implement this attack, we needed to add new capabilities to our attacker and victim:

Abilities : For the attacker in simulation

- Send plain to the victim->Using text file .
- Measure victim runtime (already done in evict and time).
- Create plaintext P [16] for a given pair i , j.
- Flush Sbox lines from cache between invocations . (In order to avoid collisions/hits from previous invocations)
- Perform "T test" to determine if a time value is considered significantly shorter than others

Abilities : For the Victim in simulation

- Load 16 bit key into cache.
- Load 256 byte Sbox into cache – 16 consecutive lines.
- Load 16 byte plaintext from file and preform the T0 table lookups for :$< P_i > \oplus < K_i > i \in \{0,4,8,12\}$ (Part of AES encryption – we won't implement the whole AES encryption since we only want to demonstrate the "first round attack"

This is how the Victim will access the T0 - table: We read the current key - then the current plain (they are in global memory in a single 16-byte block each. They will be brought to the cache after the first access).

Then we xor the plain and key and the xor result is given as an indices to the T0_sbox.

```
// now we perform the T0 table lookups for : Sbox [ key[i] xor plain[i]  ]  as i is in {0 ,4,8,12}
for(i=0;i<=12;i=i+4){
    currKey=memRead((int) &Key_arr[i]);
    currPlain=memRead((int) &plaintext[i]);
    xorRes=(currKey^currPlain);
    //Now we perform the table lookup:
    tableOutput=memRead(&T0_Sbox[xorRes]);
    printf("i=%d : curr key = %02X ,currPlain =%02X ,xorRes =%02X,tableOutput= %02X \n",i,currKey,
```

Here we can see an example for 4 iterations of the loop above. There are 3 cache misses in the first iteration, 1 in the second iteration (only the T0 accessed missed),1 in the third, and in the last iteration there was a "cache collision "with the T0_sbox access so there are no misses.

```
readByteAtAddress: cache miss:evicting set 19 ,new tag is :0x0065B
readByteAtAddress: cache miss:evicting set 20 ,new tag is :0x0065B
readByteAtAddress: cache miss:evicting set 6 ,new tag is :0x0065B
i=0 : curr key = 55 ,currPlain =66 ,xorRes =33,tableOutput= 04
readByteAtAddress:CACHE HIT ON ADDRESS 6664500 SET 19 and Tag =0x0065B
readByteAtAddress:CACHE HIT ON ADDRESS 6664516 SET 20 and Tag =0x0065B
readByteAtAddress: cache miss:evicting set 5 ,new tag is :0x0065B
i=4 : curr key = 91 ,currPlain =BB ,xorRes =2A,tableOutput= 34
readByteAtAddress:CACHE HIT ON ADDRESS 6664504 SET 19 and Tag =0x0065B
readByteAtAddress:CACHE HIT ON ADDRESS 6664520 SET 20 and Tag =0x0065B
readByteAtAddress: cache miss:evicting set 15 ,new tag is :0x0065B
i=8 : curr key = 78 ,currPlain =BB ,xorRes =C3,tableOutput= 6C
readByteAtAddress:CACHE HIT ON ADDRESS 6664508 SET 19 and Tag =0x0065B
readByteAtAddress:CACHE HIT ON ADDRESS 6664524 SET 20 and Tag =0x0065B
readByteAtAddress:CACHE HIT ON ADDRESS 6664274 SET 5 and Tag =0x0065B
i=12 : curr key = 99 ,currPlain =BB ,xorRes =22,tableOutput= B7
```

We used this 128-bit key (16 bytes) for the aes encryption - which we fixed to be:

```
void put_128_bit_key_into_global_mem (int offsetOfKeyInGlobalMem){
    //Extension : using 1 set for the key : loading 16 bytes into memory
   unsigned char key_16_bytes [16]={0x55,0x43,0xAF,0x68,0x91,0xDB,0xCC,0xE5,0x78,0xD3,0xA9,0x18,0x99,0xCE,0x6D,0xB8};
```

<u>For this key - we expect to find this information:</u>

Since We can only discover the upper half of the xor result between pairs of key bytes - we expect the attacker to discover the following (marked in yellow) information about the xor result between bytes of key.

Our key:		
index	Key byte in hex	in decimal
0	0x55	85
1	0x43	67
2	0xAF	175
3	0x68	104
4	0x91	145
5	0xDB	219
6	0xCC	204
7	0xE5	229
8	0x78	120
9	0xD3	211
10	0xA9	169
11	0x18	24
12	0x99	153
13	0xCE	206
14	0x6D	109
15	0xB8	184

Expected results: **<u>XOR BETWEEN BYTES IN KEY</u>**

Key Index	Key Index			
I	J	xor res (decimal)	xor res (hex)	only Msb
0	4	196	C4	C
0	8	45	2D	2
0	12	204	CC	C
4	8	233	E9	E
4	12	8	08	0
8	12	225	E1	E

How will the attacker construct the plaintexts that it sends to the victim:

In order to improve the success rate of the attacker - we will repeat each time measurement for I, j , k 4 times with different values for the bytes which are not tested .

Previously we suggested to fix all of the other bytes but Plain[i] and Plain[j] to 0, To reduce the influence of **"accidental" collisions** (collisions that happen because of the secret key value sending us to the same line because Key[i] xor 0 = key[i]).

If we repeat the measurement and average the times using different values for the bytes in plain, we are not testing for - we can reduce the probability of "false positives "- meaning shorter runtimes that were caused by 'accidental' collisions.

Thus, we can tell when a short runtime is a result of plain bytes i, j being set to the correct value and when it was just caused by probability (the attacker doesn't know what the key is - but it is possible that the values of the key will cause another more collisions).

How will we do it: We will run each time measurement 4 times - each time we will change the values of the bytes we are not testing for (i,j) to one of these padding values.

```
unsigned char valuesForPadding[4]={0,0xFF,0xAA,0x77};
```

For example, if we want to measure for i =0, j=4 and X =5 (xor res between Plain[i] and Plain[j]):

The plain will be (notation: **0xPP is the padded value**):

{0x05, 0xPP, 0xPP, 0xPP,0x00, 0xPP, 0xPP, 0xPP, 0xPP, 0xPP, 0xPP, 0xPP, 0xPP, 0xPP, 0xPP, 0xPP}.

The choice of P_i and P_j:

If we test for P_i and P_j we have 16 options for their xor results (as mentioned previously, we can only learn information about the MSB half of the byte of the xor result -4 bits only).

We want to test for all options: <P_i> xor <P_j> = {0, 1...15}.

So, we will always choose <P_i> (upper half of P[i]) to be the xo result) and <P_j> will always be zero (therefore <P_i> xor <P_j> = <P_i> xor 0 = <P_i> = xor result we are testing for).

Results: These are the time measurements (averaged) for the attack:

```
**********RESULTS FOR THE CACHE COLLISION **********

Time for i= 0 j= 4 is :
 2.897102  2.902908  2.771456  2.775031  2.903240  2.773237  2.777861  2.902869  2.770394  2.902619  2.902908  2.773073  2.405548  2.774528  2.916777  2.908588
Time for i= 0 j= 8 is :
 2.524374  2.526609  2.027663  2.396600  2.528884  2.524199  2.405219  2.523725  2.527967  2.524720  2.526419  2.398794  2.403093  2.528976  2.526272  2.526863
Time for i= 0 j= 12 is :
 2.898052  2.903020  2.779988  2.771478  2.904107  2.772441  2.779122  2.896760  2.777974  2.902017  2.897431  2.778134  2.401906  2.777095  2.902316  2.896618
Time for i= 4 j= 0 is :
 2.901781  2.897915  3.023546  3.028136  2.898462  3.022933  3.026434  2.904848  3.022730  2.903690  2.898150  3.027635  2.526591  3.028627  2.898254  2.897533
Time for i= 4 j= 8 is :
 2.899616  3.029957  3.022110  2.903380  3.023240  3.028122  2.900866  2.904187  3.022396  3.022571  2.898039  2.902113  2.911094  3.025711  2.527478  2.902366
Time for i= 4 j= 12 is :
 2.523000  2.900691  3.023360  2.902464  2.897481  3.028160  2.896755  3.026558  3.021997  2.901732  3.023051  2.897721  2.898439  3.025303  2.903302  3.051725
Time for i= 8 j= 0 is :
 2.538192  2.427858  2.050308  2.552609  2.420650  2.551903  2.542338  2.531403  2.536682  2.420245  2.525292  2.525150  2.530710  2.530107  2.399289  2.523919
Time for i= 8 j= 4 is :
 2.906960  2.773998  2.773199  2.904724  2.773204  2.773564  2.901675  2.898452  2.776605  2.779201  2.898381  2.897165  2.902997  2.779217  2.404404  2.901830
Time for i= 8 j= 12 is :
 2.903474  2.773004  2.780698  2.902825  2.784055  2.778345  2.897557  2.906331  2.772354  2.780647  2.898364  2.904207  2.895764  2.776194  2.404957  2.900894
Time for i= 12 j= 0 is :
 2.897592  2.897759  3.024943  3.028962  2.903513  3.024372  3.084590  2.904475  3.034967  2.902583  2.903170  3.025770  2.532443  3.023464  2.896170  2.897781
Time for i= 12 j= 4 is :
 2.521955  2.903139  3.025751  2.899977  2.905876  3.021318  2.904160  3.025572  3.022565  2.898507  3.025838  2.897681  2.904523  3.028961  2.903223  3.028988
Time for i= 12 j= 8 is :
 2.903425  3.028718  3.026413  2.900049  3.027914  3.024450  2.907824  2.898543  3.031309  3.028167  2.901784  2.897337  2.897782  3.025184  2.524341  2.901641
**********
```

To get a better view and analysis of the results, we will show them in an organized table:

		Results : time measurments for the attack :															
		Values for x (xor res for <p_i> and <p_j>)															
i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	4	2.897	2.903	2.771	2.775	2.903	2.773	2.778	2.903	2.770	2.903	2.903	2.773	2.406	2.775	2.917	2.909
0	8	2.524	2.527	2.028	2.397	2.529	2.524	2.405	2.524	2.528	2.525	2.526	2.399	2.403	2.529	2.526	2.527
0	12	2.898	2.903	2.780	2.771	2.904	2.772	2.779	2.897	2.778	2.902	2.897	2.778	2.402	2.777	2.902	2.897
4	8	2.900	3.030	3.022	2.903	3.023	3.028	2.901	2.904	3.022	3.023	2.898	2.902	2.911	3.026	2.527	2.902
4	12	2.523	2.901	3.023	2.902	2.897	3.028	2.897	3.027	3.022	2.902	3.023	2.898	2.898	3.025	2.903	3.052
8	12	2.903	2.773	2.781	2.903	2.784	2.778	2.898	2.906	2.772	2.781	2.898	2.904	2.896	2.776	2.405	2.901

We used exel to choose the **minimum value for each line** (**in green**) - which should be the correct Xor result for <K_i> xor <K_j> (Which is the information we learn with this attack).
Then we wrote down the indexes (x values which have the minimum time value in each line) and calculated the hex representation for them - and indeed the index matches the expected result:

i	j	min value	min index (hex)	differnce from average
0	4	2.405548	12 (C)	0.410585688
0	8	2.027663	2 (2)	0.436110563
0	12	2.401906	12 (C)	0.412997688
4	8	2.527478	14 (E)	0.405224875
4	12	2.523	0 (0)	0.409608688
8	12	2.404957	14 (E)	0.411272375

And the results fit the expected results for the known key (presented 2 pages ago):

					Expected result (K[i] xor K[j]	
i	j	min value	min index (hex)	differnce from average	xor res (hex)	only Msb
0	4	2.405548	12 (C)	0.410585688	C4	C
0	8	2.027663	2 (2)	0.436110563	2D	2
0	12	2.401906	12 (C)	0.412997688	CC	C
4	8	2.527478	14 (E)	0.405224875	E9	E
4	12	2.523	0 (0)	0.409608688	08	0
8	12	2.404957	14 (E)	0.411272375	E1	E

Then, we calculated the average for each line to understand if the minimum value is significantly shorter than the other time measurements. Additionally, we calculated the distance between the line average and the other time measurements to see if we got a situation of “false positive” - a wrong result that the attacker can mark as the correct one. As we can see from the results only in the right result’s column, we got a significant distance from the line’s average (several orders of magnitude larger than the other values in each line).

for the **correct xor value** - about 0.4 seconds difference from average. All of the other differences are below 0.1 seconds - so the minimum time is significantly shorter than the average and we can count on the minimum time value to accurately tell us which xor value is the correct one).

Difference of time from the average for that line																	
i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	4	0.081	0.087	0.045	0.041	0.087	0.043	0.038	0.087	0.046	0.086	0.087	0.043	0.411	0.042	0.101	0.092
0	8	0.061	0.063	0.436	0.067	0.065	0.06	0.059	0.06	0.064	0.061	0.063	0.065	0.061	0.065	0.062	0.063
0	12	0.083	0.088	0.035	0.043	0.089	0.042	0.036	0.082	0.037	0.087	0.083	0.037	0.413	0.038	0.087	0.082
4	8	0.033	0.097	0.089	0.029	0.091	0.095	0.032	0.029	0.09	0.09	0.035	0.031	0.022	0.093	0.405	0.03
4	12	0.41	0.032	0.091	0.03	0.035	0.096	0.036	0.094	0.089	0.031	0.09	0.035	0.034	0.093	0.029	0.119
8	12	0.087	0.043	0.036	0.087	0.032	0.038	0.081	0.09	0.044	0.036	0.082	0.088	0.08	0.04	0.411	0.085

This is the printout at the end of the attacker (values are in decimal):

```
<K_0> xor <K_4> = 12

<K_0> xor <K_8> = 2

<K_0> xor <K_12> = 12

<K_4> xor <K_8> = 14

<K_4> xor <K_12> = 0

<K_8> xor <K_12> = 14
**********Attacker MAIN ENDED SUCCESSFULLY
```

Summary and conclusions

- In this project we implemented/used our knowledge from multiple courses. The knowledge about Cpu cache, memory hierarchy and address decoding is from the “Computer architecture” course. The use of Unix signals and communication between processes through signals and files was learned in the “Operation systems “course. Knowledge aboud side channel attacks, AES encryption and Sboxes are from “Introduction to secure hardware “course. Knowledge about RSA encryption is from “Introduction to Cryptography”.
- In addition, we learned about the various attacks from academic papers that were linked in the assignment’s PDF as well as videos from academic courses in Ben Gurion University. We implemented the attacks according to the scenarios presented in the papers (both attacker and victim were changed for each attack scenario).
- We learned about cache side channel attacks which are a subcategory of “Architectural side channel attacks “which are using knowledge about the system architecture to create a covert /side channel that we are able to get information about the inner values and execution of a victim process.
- We implemented the simulation in c within a linux environment.
  In retrospective - this choice had advantages and disadvantages:
  Advantages:
  - C is very fast. Our simulation runs quickly - except for when we add artificial delays for cache misses.
  - C allows low level memory address manipulation through pointers - which helps when implementing the virtual global memory.
  - The process switching by using Unix signals is straightforward and was very handy in the implementation of the simulation.

  Disadvantages:
  - C doesn’t allow us to use Object oriented programing. In retrospective - the OOP approach could have helped us make the project more organized and also make the project more scalable and the code would be more readable.
  - If we used a higher-level language like java, C#, phyton - the code would have been more “portable”. The code we created can only be compiled and run on linux machines only. Porting it to windows would require a rewrite of critical parts of the simulation (i.e. signaling between processes, process runtime timing,etc).

- Debugging would have been much easier on higher level languages.

Which attack is the most powerful:

We think that the strongest attack is “flush and reload” because it is the only attack that allows us to learn the entire secret key. Other attacks are only telling us which cache lines the victim is using or reducing the complexity of brute force key guessing.

But, in order to implement this attack, one needs to have a very good understanding of the program we are attacking and also establish that the victim process and our attacking process have some form of shared memory pages, and which sets they are mapped to.

In a real system, the attacks are more complicated and require many more time measurements and statistical analysis to filter out the noise from the system and increase the certainty of the results.

How the project can be expanded in the future

- Improve the cache - create cache ways and implement various eviction policies for the cache.
- Implement system noise - by adding random amounts of delay to some of the memory reads and writes and also to the processes execution time to better simulate a real system’s behavior.
- Lower the fixed artificial “microseconds delay for cache miss “constant. We worked with a 0.5 second delay across all four attacks because it was easier to distinguish between cache hits and misses. Lowering the cache miss delay by one or 2 orders of magnitude will allow the simulation to run much faster and will be a better representation of the real world.

  According to the academic papers we read (mentioned in the theoretical chapter (FLUSH+RELOAD: a High Resolution, Low Noise -page 4) the time difference in clock cycles between a cache hit and miss is as big as in our simulation.

- Exploring and learning about various types of cache timing attacks and implement them in our simulation. In addition, comparing and grading the various types of attacks by a set of parameters:

  Runtime, number of time measurements, success rate (accuracy of results), effectiveness, harmfulness (what harm does it cause - recovering secret key or the eviction set), attacker's abilities, and how easy it is to implement.

Appendix

: all code files for attackers and victims and also the simulation methods
(fileIoFunctions.h).
The code is split between the fileIo methods which all files are using (all of the victim and attacker files) and the “fileIoFunctions.h” file is included and compiled with all of them.
Then we have separate files for each attack. For each attack we have 2 files:
Attacker_type.c, Victim_type.c so for “flush and reload the files are:
Attacker_flush.c and Victim_flush.c etc.
Attacker and Victim files might include functions that are specific to the scenario of that attack.

Cpu functions
int checkIfAddressIsIncache(int Address,char** cacheTagArray)
void UpdateCacheTextFiles(char** cacheTagArray,char ** cacheDataArray)
void waitForCacheMissDelay ()
void flush(int setToFlush,char** cacheTagArray,char** cacheDataArray)
void evictSet (int set,char *newTag ,unsigned char* newDataByteArray,char** cacheTagArray,char** cacheDataArray)
unsigned char readByteAtAddress (int address,unsigned char *globalMem,char** cacheTagArray,char** cacheDataArray)
void writeByteAtAddress (unsigned char byteTowrite,int address,unsigned char *globalMem,char** cacheTagArray,char** cacheDataArray)

Helper functions
int convertStringToNumber (char * String)
char * convertTagNumberToTagstring(int tag)
void print2dArray(char ** arr,int rows,int columns)
void free2dArray(char ** arr,int rows)
int getAddress_Set (int address)
int getAddress_Offset (int address)
int getAddress_Tag (int address)
void freeBothArrays(char** cacheTagArray,char ** cacheDataArray)
void printPlainText (int offsetOfPlainTextInGlobalMem)

File -io functions
char ** dymaciaclIyAlocateCharArr(int lines ,int columns)
char ** loadCacheDataArrayFromFile()
void writeToDataArraryFile(char ** cacheDataArray)
unsigned char* getByteArrayOfACacheDataLine(int set , char **cacheDataArray)
void writeByteArrayToACacheDataLine(int setToUpdate ,unsigned char* byteArray , char **cacheDataArray)
unsigned char getSpecficByteFromDataArray(int set,int offsetInBlock, char **cacheDataArray)

void writeByteToDataArray(unsigned char byteTowrite,int set,int offsetInBlock,char **cacheDataArray)
char * getTagAtSet (int set,char ** cacheTagArr)
void changeTagAtSet(int set,char * newTagString,char ** cacheTagArr)
char ** loadCacheTagFromFile()
void writeToTagArrayFile(char **cacheTagArray)
void UpdateCacheTextFiles(char** cacheTagArray,char ** cacheDataArray)
void writeAttackerPidToFile()
int readAttackerPidFromFile()
void writeVictimPidToFile()
int readVictimPidFromFile()
void writePlainIntoFile(unsigned char * plainTextByteArray)
void readPlaintextFromFile (unsigned char * plaintext)

Programmer level (Victim and attacker)
unsigned char memRead(int Address)
void memWrite(unsigned char ByteToWrite,int Address)
void sigUsr2(int signo) /activating -listening to the signal.

void signalOtherProccessToRun(int pid)
unsigned char ReadFromSharedMemPage(int offsetInSharedPage,char **cacheTagArray,char **cacheDataArray) //only for flush and reload

Programmer level -Victim only
Memory setup
void putSboxOntoTheGlobalMem(int offsetInsideGlobalMem,unsigned char *globalMem)
void put_128_bit_key_into_global_mem (int offsetOfKeyInGlobalMem) (cache collision only)
void put_256_byte_Sbox_In_Global_Mem (int OffsetInGlobalMem) (cache collision only)
void readPlaintextFromFile (unsigned char * plaintext)
int getPlainText_AndPutInGlobalMemory()
Cryptography methods
void Do_Some_Cryptography(int offsetOfKeyInGlobalMem,int offsetOfSboxInGlobalMem) // simple xor between plain and key. (used in prime & probe, evict & time)
void Extended_Do_Some_Cryptography(int offsetOfKeyInGlobalMem,int offsetOfSboxInGlobalMem) //// simple xor between plain and key-key is 16 bytes. (used in evict & time)
void perform_AES_first_round (int offsetOfPlainInGlobalMem ,int offsetOfSboxInGlobalMem,int OffsetOfKeyInGlobalMem) (cache collision only)

Programmer level -Attacker only
int prime_Specific_Set (int specificSet) //for prime and probe
double measureTimeToReadFromAddress(int address) //for prime & probe , flush & reload
double measureOtherProcessRuntime(int victimPid) // for evict & time , cache collision(slightly different)

void changePlaintext(int i,int j,int x,unsigned char ValueForPadding,unsigned char *plain_16_bytes) // for cache collision only
void changePlaintext(int i,int j,int x,unsigned char ValueForPadding,unsigned char *plain_16_bytes) // for cache collision only
void flushSboxLinesFromCache (int setSboxStartsAt , int setSboxEndsAt) // for cache collision only

FIRST ATTACK - PRIME AND PROBE

Victim_prime.c:

```
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <fcntl.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include "fileIOFunctions.h"
#include <math.h>
#define global_mem_size 65536
#define number_Of_Hex_Chars_For_Tag 5
#define number_Of_Hex_Chars_For_Set 2
#define number_Of_Hex_Chars_For_Offset 1
#define number_Of_Lines_In_Cache 256
#define micro_seconds_delay_for_cache_miss 500000 //half a second

//global arrays for the programmer level
unsigned char * globalMem ;
int pointerTounAllocatedMemory=0;
char** cacheTagArray;
char** cacheDataArray;

//Simple memory read/write methods for the programmer
unsigned char memRead(int Address){
      return readByteAtAddress(Address,globalMem,cacheTagArray,cacheDataArray);
}

void memWrite(unsigned char ByteToWrite,int Address){
      writeByteAtAddress(ByteToWrite,Address,
globalMem,cacheTagArray,cacheDataArray);
}

//signal function to invoke the victim to run:
void sigUsr2(int signo){ //code =12
      signal(SIGUSR2,sigUsr2);//reloading for backwards linux compatibility
}

void Do_Some_Cryptography(int offsetOfKeyInGlobalMem,int
offsetOfSboxInGlobalMem){
      unsigned char randomPlainText[16];
      //reading key:
   // printf("offsetOfSboxInGlobalMem= %d\n",offsetOfSboxInGlobalMem);
```

```
	unsigned char key=memRead((int)&globalMem[offsetOfKeyInGlobalMem]);
	int i;
	//generating random plainText.
	for(i=0;i<16;i++){
	randomPlainText[i]=rand()%16;
	}
	//i ,key ,xorRes are in a register. randomPlainText isn't stored in
memory.
	//now we encrypt : SBOX(key xor plain)
	//unsigned char cipher
[16]={0x00,0x00,0x00,0x00,0x00,0x0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
,0x00};
	//assaign space for the cipher in global mem:
	int OffsetOfCipherInMemory=pointerTounAllocatedMemory;
	pointerTounAllocatedMemory+=16;
    for(i=0;i<16;i++){
	unsigned char xorRes=(key^randomPlainText[i])%16;
	unsigned char Cipher= memRead(&globalMem[offsetOfSboxInGlobalMem+xorRes])
;
	memWrite(Cipher,&globalMem[OffsetOfCipherInMemory+i]);
	}
	//release cipher memory after calculation is finished (it is a local
array to the function):
	pointerTounAllocatedMemory+=-16;
}

//we let control of the cpu to the other process
// before we do that we need to update the cache files with current state.
void signalOtherProccessToRun(int pid ){
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	kill(pid,SIGUSR2);
}


int main(){
	writeVictimPidToFile();
	//loading the signal sigusr2
	signal(SIGUSR2,sigUsr2);
	//defining the global memory space (in DRAM) for the process:
	globalMem=(unsigned char *)malloc(global_mem_size*(sizeof(unsigned
char)));
	int offsetOfSboxInGlobalMem=pointerTounAllocatedMemory;
	//Hard coded sbox is in Dram:

putSboxOntoTheGlobalMem(offsetOfSboxInGlobalMem,&globalMem[offsetOfSboxInGlobal
Mem]);
	pointerTounAllocatedMemory+=16;
	//put secret Key =0xEA into memory in an arbitrary location
	int
offsetOfKeyInGlobalMem=rand()%(global_mem_size-pointerTounAllocatedMemory)+poin
terTounAllocatedMemory;
	globalMem[offsetOfKeyInGlobalMem]=0xEA;
	pointerTounAllocatedMemory+=16;
	//Victim waits to be called invoked by the system/other process.Runs some
cryptography and then waits again.
	while(1){
```

```
	pause();//wainting for sigusr2 signal from the system (the attacker can
invoke it to run)
	int callingProcessPid =readAttackerPidFromFile();
	printf("Victim is invoked and running :\n");
	printf("Doing some symmetric cryptography:\n");
	//loading cache state:
	cacheTagArray = loadCacheTagFromFile();
	cacheDataArray=loadCacheDataArrayFromFile();
	Do_Some_Cryptography(offsetOfKeyInGlobalMem,offsetOfSboxInGlobalMem);
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	signalOtherProccessToRun(callingProcessPid);
	printf("******victim ended -waiting for another invocation******\n");
	}
	free (globalMem);
	return 0;
}
```

Attacker_prime.c

```
#include "fileIOFunctions.h"
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <fcntl.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>
#define global_mem_size 65536
#define number_Of_Hex_Chars_For_Tag 5
#define number_Of_Hex_Chars_For_Set 2
#define number_Of_Hex_Chars_For_Offset 1
#define number_Of_Lines_In_Cache 256
#define micro_seconds_delay_for_cache_miss 500000 //half a second

//global arrays for the programmer level
unsigned char * globalMem ;
int pointerTounAllocatedMemory=0;
char** cacheTagArray;
char** cacheDataArray;
```

```
//Simple memory read/write methods for the programmer
unsigned char memRead(int Address){
	return readByteAtAddress(Address,globalMem,cacheTagArray,cacheDataArray);
}

void memWrite(unsigned char ByteToWrite,int Address){
	writeByteAtAddress(ByteToWrite,Address,
globalMem,cacheTagArray,cacheDataArray);
}

//signal function that the victim process can use to tell us that it has
finished processing/waiting and it's our turn to run
void sigUsr2(int signo){ //code =12
	signal(SIGUSR2,sigUsr2);//reloading for backwards linux compatibility
}


//Primes a specific set and also returns the address which was used to prime
the set
//(that we read/wrote to in order to bring it to cache)
int prime_Specific_Set (int specificSet){
	//firstly find out which set is the Global Memory starting at:
	int addressOfTheBeginingOfVirtualMemory=globalMem;
	int currSet =getAddress_Set(addressOfTheBeginingOfVirtualMemory);
	//calculate how many 16 byte blocks - to advance in order to get to the
set:
	int Differnce=0;
	if(specificSet>currSet){
	Differnce=specificSet-currSet;
	}
	if(specificSet<currSet){
	Differnce=256-currSet;
	Differnce=Differnce+specificSet;
	}
	int
AddressWithThe_specificSet=addressOfTheBeginingOfVirtualMemory+16*Differnce;
	//Now ,we that have the address - we can "prime" the cache set by
reading/Writing from/to the address:
	memWrite(0xFF,AddressWithThe_specificSet);
	return AddressWithThe_specificSet;
}

//Reads from that address and returns the time it took to complete the read
double measureTimeToReadFromAddress(int address){
	//setup:
	struct timeval  startTime, endTime;
	gettimeofday(&startTime, NULL);
	//Reading:
	memRead(address);
	gettimeofday(&endTime, NULL);
	double time=(double) (endTime.tv_usec - startTime.tv_usec) / 1000000 +
(double) (endTime.tv_sec - startTime.tv_sec);
   // printf ("Total time = %f seconds\n",time);
	return time;

}
```

```
//we let control of the cpu to the other process
// before we do that we need to update the cache files with current state.
void signalOtherProccessToRun(int pid ){
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	kill(pid,SIGUSR2);
}



int main(){
	//loading the signal sigusr2
	signal(SIGUSR2,sigUsr2);
	cacheTagArray = loadCacheTagFromFile();
	cacheDataArray=loadCacheDataArrayFromFile();
	//defining the global memory space (in DRAM) for this process:
	globalMem=(unsigned char *)malloc(global_mem_size*(sizeof(unsigned
char)));
	writeAttackerPidToFile();
	int victimPid =readVictimPidFromFile();
	double timeToAccessSet_Before[256];
	double timeToAcssesSet_After [256];
	int i=0;
	//we evict all of the sets -one by one - and then we invoke the victim
to run .
	//then we try to read the data again - and measure the time it took to
read.
	//(1) prime  (2) measure access time before (3) let victim run (4)
measure access time after
	for(i=0;i<256;i++){
	int addressWeUsed= prime_Specific_Set(i); //(1)
	  timeToAccessSet_Before[i]=measureTimeToReadFromAddress(addressWeUsed);
//(2)
	signalOtherProccessToRun(victimPid); //(3)
	pause();//waits for signal from victim to proceed execution.
	//we reload from files to get the updated cache state:
	cacheTagArray = loadCacheTagFromFile();
	cacheDataArray=loadCacheDataArrayFromFile();
	//(4)
	timeToAcssesSet_After[i]=measureTimeToReadFromAddress(addressWeUsed);
	}

	//We  finished the attack. Now we can print the results :
	printf("    time before:            time after: \n");

	for(i=0;i<256;i++) {
	printf("set_%d :%f
%f\n",i,timeToAccessSet_Before[i],timeToAcssesSet_After[i] );
    }



	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	printf("******************Attacker MAIN ENDED SUCCESSFULLY
!**********************\n");
	free(globalMem);
	return 0;
}
```

Flush and Reload
Victim_flush.c

```
#include "fileIOFunctions.h"
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <fcntl.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>
#define global_mem_size 65536
#define number_Of_Hex_Chars_For_Tag 5
#define number_Of_Hex_Chars_For_Set 2
#define number_Of_Hex_Chars_For_Offset 1
#define number_Of_Lines_In_Cache 256
#define micro_seconds_delay_for_cache_miss 500000 //half a second



//Note - if this file does not compile it is because the compiler doesn't
recognize the pow() instruction.
//In order to compile this file you have to use this terminal command:
//  gcc -o Victim_flush  Victim_flush.c -lm

//global arrays for the programmer level
unsigned char * globalMem ;
int pointerTounAllocatedMemory=0;
char** cacheTagArray;
char** cacheDataArray;

//Simple memory read/write methods for the programmer
unsigned char memRead(int Address){
      return readByteAtAddress(Address,globalMem,cacheTagArray,cacheDataArray);
}

void memWrite(unsigned char ByteToWrite,int Address){
      writeByteAtAddress(ByteToWrite,Address,
globalMem,cacheTagArray,cacheDataArray);
}

//signal function to invoke the victim to run:
void sigUsr2(int signo){ //code =12
      signal(SIGUSR2,sigUsr2);//reloading for backwards linux compatibility
```

```
}



putMultplyAndAddInstructionsInGlobalMem(int offsetInsideGlobalMem,unsigned char
*globalMem){
	// We are loading a specific memory (and cache) block with "Multiply and
Add" commands (part of RSA execution).
	//this instructions will be accessed by the program if they are executed
during RSA encryption.
	int i=0;
	unsigned char instructionsOpcodes[16]
={0x56,0xDD,0x1F,0x71,0x13,0x26,0x31,0x44,0xEB,0x9B,0x5F,0x05,0x29,0x8A,0x0A,0x
10};
	//copying into global mem
	for(i=0;i<16;i++){
	globalMem[offsetInsideGlobalMem+i]=instructionsOpcodes[i];
	}
}

//the shared page only has the instructionsOpcodes for  MultplyAndAdd
//offset is range 0-15 (inclusive).
unsigned char ReadFromSharedMemPage(int offsetInSharedPage,char
**cacheTagArray,char **cacheDataArray){
	int simulated_Location_Of_page=0x00002640; //maps to set 100 . tag =00002
.offset 0 (aligned)
	unsigned char instructionsOpcodes[16]
={0x56,0xDD,0x1F,0x71,0x13,0x26,0x31,0x44,0xEB,0x9B,0x5F,0x05,0x29,0x8A,0x0A,0x
10};
	int
isAddressAlreadyIncache=checkIfAddressIsIncache(simulated_Location_Of_page,cach
eTagArray);
	int set=getAddress_Set(simulated_Location_Of_page);
	//if it is not currently in cache - we are evicting line 100 - and
storing instructionsOpcodes[] in cache:
	if(isAddressAlreadyIncache==0){
	printf("cache miss on ReadFromSharedMemPage Address %d .evicting set
%d\n" ,simulated_Location_Of_page+offsetInSharedPage,set);
	waitForCacheMissDelay(); //cache miss - 0.5 seconds delay
	int numerical_tag =getAddress_Tag(simulated_Location_Of_page);
	char *tagStr=convertTagNumberToTagstring(numerical_tag);//dynamic -should
be freed
	evictSet(set,tagStr,instructionsOpcodes,cacheTagArray,cacheDataArray);
	free(tagStr);
	}
	else{
	printf("cache HIT on ReadFromSharedMemPage Address %d . set %d\n"
,simulated_Location_Of_page+offsetInSharedPage,set);
	}
	//reading the byte from cache and returning it
	unsigned char byte
=getSpecficByteFromDataArray(set,offsetInSharedPage,cacheDataArray);
	return byte;

}
```

```
//we let control of the cpu to the other process
// before we do that we need to update the cache files with current state.
void signalOtherProccessToRun(int pid ){
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	kill(pid,SIGUSR2);
}


int main(){
	writeVictimPidToFile();
	//loading the signal sigusr2
	signal(SIGUSR2,sigUsr2);
	//defining the global memory space (in DRAM) for the process:
	globalMem=(unsigned char *)malloc(global_mem_size*(sizeof(unsigned
char)));
	int offsetOfSboxInGlobalMem=pointerTounAllocatedMemory;
	//Hard coded sbox is in Dram:

putSboxOntoTheGlobalMem(offsetOfSboxInGlobalMem,&globalMem[offsetOfSboxInGlobal
Mem]);
	pointerTounAllocatedMemory+=16;
	//put secret Key =0xEA into memory in an arbitrary location
	int
offsetOfKeyInGlobalMem=rand()%(global_mem_size-pointerTounAllocatedMemory)+poin
terTounAllocatedMemory;
	globalMem[offsetOfKeyInGlobalMem]=0xEA;
	pointerTounAllocatedMemory+=16;

	//We Are loading the multiply and add are in shared memory in a location
that corresponds to set 100.
	//We will put them in an arbitrary location which corresponds to set 100:
	cacheTagArray = loadCacheTagFromFile();
	cacheDataArray=loadCacheDataArrayFromFile();
	//Victim waits to be called invoked by the system/other process.Runs some
cryptography and then waits again.
	//waiting for the attacker to call for the sta
	int private_SecretKey_d=0xFFABCD91;// (exponent e) - 32 bit
	int m=256;//modulu for exponent function .
	int b=0x2A771308;  //the number we calculate b^d =? (cipher to decrypt)
	int x=1 ; //result for exponent

	pause();// waiting for the other process (Attacker) to ask for decryption
	int callingProcessPid =readAttackerPidFromFile();
	cacheTagArray = loadCacheTagFromFile();
	cacheDataArray=loadCacheDataArrayFromFile();
	printf("Victim is invoked to run RSA decryption :\n");
	int i;
	//exponent(b,(d=e),m ):
	for( i=31;i>=0;i--){ //exponent is 32 bits long
	x=pow((double)x,(double)2);
	x=x%m;
	int d_i=getSpecificBitFromInteger(private_SecretKey_d,i);
	if(d_i==1){
		//We excecute the if->we are accessing the line in shared memory -
		//and the block will be read and loaded into cache
```

```
            ReadFromSharedMemPage(0,cacheTagArray,cacheDataArray);
            x=x*b;
            x=x%m;
      }
      //at the end of every iteration we are letting the other
process(attacker) the control over the cpu.
      UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
      printf("Victim has ended iteration %d\n",i);
      signalOtherProccessToRun(callingProcessPid);
      pause(); //waiting to get the cpu again -before continuing to next
iteration
      //loading cache state
      int nextI=i-1;
      printf("Victim is resuming for iteration %d\n",nextI);
      cacheTagArray = loadCacheTagFromFile();
      cacheDataArray=loadCacheDataArrayFromFile();

      }


      free (globalMem);
      return 0;
}
```

Attacker_flush.c

```
#include "fileIOFunctions.h"
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <fcntl.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>
#define global_mem_size 65536
#define number_Of_Hex_Chars_For_Tag 5
#define number_Of_Hex_Chars_For_Set 2
#define number_Of_Hex_Chars_For_Offset 1
#define number_Of_Lines_In_Cache 256
#define micro_seconds_delay_for_cache_miss 500000 //half a second?

//global arrays for the programmer level
unsigned char * globalMem ;
int pointerTounAllocatedMemory=0;
char** cacheTagArray;
char** cacheDataArray;

//Simple memory read/write methods for the programmer
unsigned char memRead(int Address){
      return readByteAtAddress(Address,globalMem,cacheTagArray,cacheDataArray);
```

```
}

void memWrite(unsigned char ByteToWrite,int Address){
	writeByteAtAddress(ByteToWrite,Address,
globalMem,cacheTagArray,cacheDataArray);
}

//signal function that the victim process can use to ell us that it has
finished processing/waiting and it's our turn to run
void sigUsr2(int signo){ //code =12
	signal(SIGUSR2,sigUsr2);//reloading for backwards linux compatibility
}



//the shared page only has the instructionsOpcodes for  MultplyAndAdd
//offset is range 0-15 (inclusive).
unsigned char ReadFromSharedMemPage(int offsetInSharedPage,char
**cacheTagArray,char **cacheDataArray){
	int simulated_Location_Of_page=0x00002640; //maps to set 100 . tag =00002
.offset 0 (aligned)
	unsigned char instructionsOpcodes[16]
={0x56,0xDD,0x1F,0x71,0x13,0x26,0x31,0x44,0xEB,0x9B,0x5F,0x05,0x29,0x8A,0x0A,0x
10};
	int
isAddressAlreadyIncache=checkIfAddressIsIncache(simulated_Location_Of_page,cach
eTagArray);
	int set=getAddress_Set(simulated_Location_Of_page);
	//if it is not currently in cache - we are evicting line 100 - and
storing instructionsOpcodes[] in cache:
	if(isAddressAlreadyIncache==0){
	printf("cache miss on ReadFromSharedMemPage Address %d .evicting set
%d\n" ,simulated_Location_Of_page+offsetInSharedPage,set);
	waitForCacheMissDelay(); //cache miss - 0.5 seconds delay
	int numerical_tag =getAddress_Tag(simulated_Location_Of_page);
	char *tagStr=convertTagNumberToTagstring(numerical_tag);//dynamic -should
be freed
	evictSet(set,tagStr,instructionsOpcodes,cacheTagArray,cacheDataArray);
	free(tagStr);
	}
	else{
	printf("cache HIT on ReadFromSharedMemPage Address %d . set %d\n"
,simulated_Location_Of_page+offsetInSharedPage,set);
	}
	//reading the byte from cache and returning it
	unsigned char byte
=getSpecficByteFromDataArray(set,offsetInSharedPage,cacheDataArray);
	return byte;

}


//Reads from that address and returns the time it took to complete the read
double measureTimeToReadFromAddress(int address){
	//setup:
	struct timeval  startTime, endTime;
	gettimeofday(&startTime, NULL);
	//Reading -using the ReadFromSharedMemPage instead of memRead() - because
we are reading from shared memory address in this specific attack:
	ReadFromSharedMemPage(0,cacheTagArray,cacheDataArray);
	//memRead(address);
	gettimeofday(&endTime, NULL);
	double time=(double) (endTime.tv_usec - startTime.tv_usec) / 1000000 +
(double) (endTime.tv_sec - startTime.tv_sec);
   // printf ("Total time = %f seconds\n",time);
	return time;
}


//we let control of the cpu to the other process
// before we do that we need to update the cache files with the current state.
```

```
void signalOtherProccessToRun(int pid ){
      UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
      kill(pid,SIGUSR2);
}


int main(){
    //loading the signal sigusr2
      signal(SIGUSR2,sigUsr2);
      cacheTagArray = loadCacheTagFromFile();
      cacheDataArray=loadCacheDataArrayFromFile();
      //defining the global memory space (in DRAM) for this process:
      globalMem=(unsigned char *)malloc(global_mem_size*(sizeof(unsigned
char)));
      writeAttackerPidToFile();
      int victimPid =readVictimPidFromFile();
      double timeToAccess_criticalLines[32];
      int i=0;
      //we flush all of the sets -one by one - and then we invoke the victim to
run .
      //then we try to read the data again - and measure the time it took to
read.
      //(1) flush  (2) wait for other process to run (3) Reloading and
measuring time
      int sharedAddress=0x00002640;
      int setToflush =getAddress_Set(sharedAddress); //The attacker knows the
shared address and it's set.
      for(i=31;i>=0;i--){
      flush(setToflush,cacheTagArray,cacheDataArray); //((1)
      //now we let the victim to run:(2)
      UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
      signalOtherProccessToRun(victimPid);
      pause();//waiting for the victim to return the control to us.
          //Getting the updated cache state:
      cacheTagArray = loadCacheTagFromFile();
      cacheDataArray=loadCacheDataArrayFromFile();
      //(3) Reloading

timeToAccess_criticalLines[i]=measureTimeToReadFromAddress(sharedAddress);
      }



      //We  finished the attack. Now we can print the results :
      printf("\n****************The results are*********************  \n");
      int guessedBits [32];
      int RecoveredKey =0;
      for(i=31;i>=0;i--) {
      printf("Access time on iteration %d is %f ",32-i,
timeToAccess_criticalLines[i]);
      int didWeHaveAshortAccessTime=timeToAccess_criticalLines[i]<0.5;
      if(didWeHaveAshortAccessTime)  //short time ->cache hit ->the victim
accessed the critical lines ->d_i was 1.
            guessedBits[i]=1;
      else                                 //long time ->cache miss ->victim
did not accesses critical lines-> d_i was 0.
            guessedBits[i]=0;
      printf("guessed bit =%d \n", guessedBits[i]);
      RecoveredKey+=(guessedBits[i]<<i) ; //calculating the decimal value of
the key : adding bit*2^weight

      }

      printf("guessedKeyInBinary: ");
      for(i=31;i>=0;i--){
      printf("%d",guessedBits[i]);
      }
      printf("\n");
      printf("RecoveredKey in decimal %d . in hex :%X
\n",RecoveredKey,RecoveredKey);
```

```
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	printf("******************Attacker MAIN ENDED SUCCESSFULLY
!**********************\n");
	free(globalMem);
	return 0;
}
```

Victim_evict_time.c

```
#include "fileIOFunctions.h"
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <fcntl.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>
#define global_mem_size 65536
#define number_Of_Hex_Chars_For_Tag 5
#define number_Of_Hex_Chars_For_Set 2
#define number_Of_Hex_Chars_For_Offset 1
#define number_Of_Lines_In_Cache 256
#define micro_seconds_delay_for_cache_miss 500000 //half a second

//global arrays for the programmer level
unsigned char * globalMem ;
int pointerTounAllocatedMemory=0;
char** cacheTagArray;
char** cacheDataArray;

//Simple memory read/write methods for the programmer
unsigned char memRead(int Address){
	return readByteAtAddress(Address,globalMem,cacheTagArray,cacheDataArray);
}

void memWrite(unsigned char ByteToWrite,int Address){
	writeByteAtAddress(ByteToWrite,Address,
globalMem,cacheTagArray,cacheDataArray);
}

//signal function to invoke the victim to run:
void sigUsr2(int signo){ //code =12
	signal(SIGUSR2,sigUsr2);//reloading for backwards linux compatibility
}



putMultplyAndAddInstructionsInGlobalMem(int offsetInsideGlobalMem,unsigned char
*globalMem){
	// We are loading a specific memory (and cache) block with "Multiply and
Add" commands (part of RSA execution).
	//this instructions will be accessed by the program if they are executed
during RSA encryption.
	int i=0;
	unsigned char instructionsOpcodes[16]
={0x56,0xDD,0x1F,0x71,0x13,0x26,0x31,0x44,0xEB,0x9B,0x5F,0x05,0x29,0x8A,0x0A,0x
10};
	//copying into global mem
	for(i=0;i<16;i++){
	globalMem[offsetInsideGlobalMem+i]=instructionsOpcodes[i];
	}
```

```
}

//the shared page only has the instructionsOpcodes for  MultplyAndAdd
//offset is range 0-15 (inclusive).
unsigned char ReadFromSharedMemPage(int offsetInSharedPage,char
**cacheTagArray,char **cacheDataArray){
	int simulated_Location_Of_page=0x00002640; //maps to set 100 . tag =00002
.offset 0 (aligned)
	unsigned char instructionsOpcodes[16]
={0x56,0xDD,0x1F,0x71,0x13,0x26,0x31,0x44,0xEB,0x9B,0x5F,0x05,0x29,0x8A,0x0A,0x
10};
	int
isAddressAlreadyIncache=checkIfAddressIsIncache(simulated_Location_Of_page,cach
eTagArray);
	int set=getAddress_Set(simulated_Location_Of_page);
	//if it is not currently in cache - we are evicting line 100 - and
storing instructionsOpcodes[] in cache:
	if(isAddressAlreadyIncache==0){
	printf("cache miss on ReadFromSharedMemPage Address %d .evicting set
%d\n" ,simulated_Location_Of_page+offsetInSharedPage,set);
	waitForCacheMissDelay(); //cache miss - 0.5 seconds delay
	int numerical_tag =getAddress_Tag(simulated_Location_Of_page);
	char *tagStr=convertTagNumberToTagstring(numerical_tag);//dynamic -should
be freed
	evictSet(set,tagStr,instructionsOpcodes,cacheTagArray,cacheDataArray);
	free(tagStr);
	}
	else{
	printf("cache HIT on ReadFromSharedMemPage Address %d . set %d\n"
,simulated_Location_Of_page+offsetInSharedPage,set);
	}
	//reading the byte from cache and returning it
	unsigned char byte
=getSpecficByteFromDataArray(set,offsetInSharedPage,cacheDataArray);
	return byte;

}

void Do_Some_Cryptography(int offsetOfKeyInGlobalMem,int
offsetOfSboxInGlobalMem){
	unsigned char randomPlainText[16];
	//reading key:
	unsigned char key=memRead((int)&globalMem[offsetOfKeyInGlobalMem]);
	int i;
	//generating random plainText.
	for(i=0;i<16;i++){
	randomPlainText[i]=rand()%16;
	}
	//i ,key ,xorRes are in a register. randomPlainText isn't stored in
memory.
	//now we encrypt : SBOX(key xor plain)
	//unsigned char cipher
[16]={0x00,0x00,0x00,0x00,0x00,0x0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
,0x00};
	//assign space for the cipher in global mem:
	int OffsetOfCipherInMemory=pointerTounAllocatedMemory;
	pointerTounAllocatedMemory+=16;
	for(i=0;i<16;i++){
	unsigned char xorRes=(key^randomPlainText[i])%16;
	unsigned char Cipher= memRead(&globalMem[offsetOfSboxInGlobalMem+xorRes])
;
	memWrite(Cipher,&globalMem[OffsetOfCipherInMemory+i]);
	}
	//release cipher memory after calculation is finished (it is a local
array to the function):
	pointerTounAllocatedMemory+=-16;
}

void Extended_Do_Some_Cryptography(int offsetOfKeyInGlobalMem,int
offsetOfSboxInGlobalMem){
	unsigned char randomPlainText[16];
```

```
	//reading key:
	//Extension to 16 blocks of key !
	int i;
	unsigned char key[16];
	for(i=0;i<16;i++){
	key[i]=memRead((int)&globalMem[offsetOfKeyInGlobalMem]);
	offsetOfKeyInGlobalMem+=16;

	}
	//generating random plainText.
	for(i=0;i<16;i++){
	randomPlainText[i]=rand()%16;
	}
	//i ,key ,xorRes are in a register. randomPlainText isn't stored in
memory.
	//now we encrypt : SBOX(key xor plain)
	//unsigned char cipher
[16]={0x00,0x00,0x00,0x00,0x00,0x0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
,0x00};
	//assign space for the cipher in global mem:
	int OffsetOfCipherInMemory=pointerTounAllocatedMemory;
	pointerTounAllocatedMemory+=16;
	for(i=0;i<16;i++){
	//Extension to use 16 byte key:
	unsigned char xorRes=(key[i]^randomPlainText[i])%16;
	unsigned char Cipher= memRead(&globalMem[offsetOfSboxInGlobalMem+xorRes])
;
	   memWrite(Cipher,&globalMem[OffsetOfCipherInMemory+i]);
	}
	//release cipher memory after calculation is finished (it is a local
array to the function):
	pointerTounAllocatedMemory+=-16;
}


//we let control of the cpu to the other process
// before we do that we need to update the cache files with current state.
void signalOtherProccessToRun(int pid ){
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	kill(pid,SIGUSR2);
}

int main(){
	writeVictimPidToFile();
	//loading the signal sigusr2
	signal(SIGUSR2,sigUsr2);
	//defining the global memory space (in DRAM) for the process:
	globalMem=(unsigned char *)malloc(global_mem_size*(sizeof(unsigned
char)));
	int offsetOfSboxInGlobalMem=pointerTounAllocatedMemory;
	//Hard coded sbox is in Dram:

putSboxOntoTheGlobalMem(offsetOfSboxInGlobalMem,&globalMem[offsetOfSboxInGlobal
Mem]);
	pointerTounAllocatedMemory+=16;
	//put secret Key =0xEA into memory in an arbitrary location
	int
offsetOfKeyInGlobalMem=rand()%(global_mem_size-pointerTounAllocatedMemory)+poin
terTounAllocatedMemory;
	globalMem[offsetOfKeyInGlobalMem]=0xEA;
	pointerTounAllocatedMemory+=16;

	//Extension : using more sets : loading 15 more blocks into the
	unsigned char moreKeys
[15]={0x55,0x43,0xAF,0x68,0x91,0xDB,0xCC,0xE5,0x48,0x03,0x26,0x31,0x69,0x88,0xB
B};
	int i;
	int Running_offsetOfKeyInGlobalMem=offsetOfKeyInGlobalMem;
    for(i=0;i<15;i++){
	Running_offsetOfKeyInGlobalMem+=16;
	pointerTounAllocatedMemory+=16;
```

```
        globalMem[Running_offsetOfKeyInGlobalMem]=moreKeys[i];
        }


        cacheTagArray = loadCacheTagFromFile();
        cacheDataArray=loadCacheDataArrayFromFile();
        //Victim waits to be called invoked by the system/other process.Runs some
cryptography and then waits again.
        //waiting for the attacker to call for the sta
        int counter=0;

    while(1){
        pause();//waiting for sigusr2 signal from the system (the attacker can
invoke it to run)
        int callingProcessPid =readAttackerPidFromFile();
        counter++;
        printf("Victim is invoked and running for the %d time  :\n",counter);
        printf("Doing some symetric cryptography:\n");
        //loading cache state:
        cacheTagArray = loadCacheTagFromFile();
        cacheDataArray=loadCacheDataArrayFromFile();
        //Do_Some_Cryptography(offsetOfKeyInGlobalMem,offsetOfSboxInGlobalMem);

Extended_Do_Some_Cryptography(offsetOfKeyInGlobalMem,offsetOfSboxInGlobalMem);
        UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
        signalOtherProccessToRun(callingProcessPid);
        printf("******victim ended -waiting for another invocation******\n");
        }
        free (globalMem);
        return 0;
}
```

Attacker_evict_time.c

```
#include "fileIOFunctions.h"
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <fcntl.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>
#define global_mem_size 65536
#define number_Of_Hex_Chars_For_Tag 5
#define number_Of_Hex_Chars_For_Set 2
#define number_Of_Hex_Chars_For_Offset 1
#define number_Of_Lines_In_Cache 256
#define micro_seconds_delay_for_cache_miss 500000 //half a second
```

```
//global arrays for the programmer level
unsigned char * globalMem ;
int pointerTounAllocatedMemory=0;
char** cacheTagArray;
char** cacheDataArray;

//Simple memory read/write methods for the programmer
unsigned char memRead(int Address){
	return readByteAtAddress(Address,globalMem,cacheTagArray,cacheDataArray);
}

void memWrite(unsigned char ByteToWrite,int Address){
	writeByteAtAddress(ByteToWrite,Address,
globalMem,cacheTagArray,cacheDataArray);
}

//signal function that the victim process can use to ell us that it has
finished processing/waiting and it's our turn to run
void sigUsr2(int signo){ //code =12
	signal(SIGUSR2,sigUsr2);//reloading for backwards linux compatibility
}


//Primes a specific set and also returns the address which was used to prime
the set
//(that we read/wrote to in order to bring it to cache)
int prime_Specific_Set (int specificSet){
	//firstly find out which set is the Global Memory starting at:
	int addressOfTheBeginingOfVirtualMemory=globalMem;
	int currSet =getAddress_Set(addressOfTheBeginingOfVirtualMemory);
	//calculate how many 16 byte blocks - to advance in order to get to the
set:
	int Differnce=0;
	if(specificSet>currSet){
	Differnce=specificSet-currSet;
	}
	if(specificSet<currSet){
	Differnce=256-currSet;
	Differnce=Differnce+specificSet;
	}
	int
AddressWithThe_specificSet=addressOfTheBeginingOfVirtualMemory+16*Differnce;
	//Now ,we that have the address - we can "prime" the cache set by
reading/Writing from/to the address:
	memWrite(0xFF,AddressWithThe_specificSet);
	return AddressWithThe_specificSet;
}


//the shared page only has the instructionsOpcodes for  MultplyAndAdd
//offset is range 0-15 (inclusive).
unsigned char ReadFromSharedMemPage(int offsetInSharedPage,char
**cacheTagArray,char **cacheDataArray){
	int simulated_Location_Of_page=0x00002640; //maps to set 100 . tag =00002
.offset 0 (aligned)
```

```
	unsigned char instructionsOpcodes[16]
={0x56,0xDD,0x1F,0x71,0x13,0x26,0x31,0x44,0xEB,0x9B,0x5F,0x05,0x29,0x8A,0x0A,0x
10};
	int
isAddressAlreadyIncache=checkIfAddressIsIncache(simulated_Location_Of_page,cach
eTagArray);
	int set=getAddress_Set(simulated_Location_Of_page);
	//if it is not currently in cache - we are evicting line 100 - and
storing instructionsOpcodes[] in cache:
	if(isAddressAlreadyIncache==0){
	printf("cache miss on ReadFromSharedMemPage Address %d .evicting set
%d\n" ,simulated_Location_Of_page+offsetInSharedPage,set);
	waitForCacheMissDelay(); //cache miss - 0.5 seconds delay
	int numerical_tag =getAddress_Tag(simulated_Location_Of_page);
	char *tagStr=convertTagNumberToTagstring(numerical_tag);//dynamic -should
be freed
	evictSet(set,tagStr,instructionsOpcodes,cacheTagArray,cacheDataArray);
	free(tagStr);
	}
	else{
	printf("cache HIT on ReadFromSharedMemPage Address %d . set %d\n"
,simulated_Location_Of_page+offsetInSharedPage,set);
	}
	//reading the byte from cache and returning it
	unsigned char byte
=getSpecficByteFromDataArray(set,offsetInSharedPage,cacheDataArray);
	return byte;

}



//measure Runtime of the victim process -by signaling it and waiting for it to
signal back to us .
double measureOtherProcessRuntime(int victimPid){
	//setup:
	struct timeval  startTime, endTime;
	gettimeofday(&startTime, NULL);
	//5 lines for process switching:
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	signalOtherProccessToRun(victimPid); //letting the victim run
	pause(); //wating for victim to finish
	cacheTagArray = loadCacheTagFromFile();
	cacheDataArray=loadCacheDataArrayFromFile();
	gettimeofday(&endTime, NULL);
	double time=(double) (endTime.tv_usec - startTime.tv_usec) / 1000000 +
(double) (endTime.tv_sec - startTime.tv_sec);
	return time;

}


//we let control of the cpu to the other process
// before we do that we need to update the cache files with the current state.
void signalOtherProccessToRun(int pid ){
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	kill(pid,SIGUSR2);
```

```
}


int main(){
	//loading the signal sigusr2
	signal(SIGUSR2,sigUsr2);
	cacheTagArray = loadCacheTagFromFile();
	cacheDataArray=loadCacheDataArrayFromFile();
	//defining the global memory space (in DRAM) for this process:
	globalMem=(unsigned char *)malloc(global_mem_size*(sizeof(unsigned
char)));
	writeAttackerPidToFile();
	int victimPid =readVictimPidFromFile();
	double excutionTimeAfterEvicting[256];
	int i=0;
	//Firstly - we average the victim runtime before we start 'messing' with
the cache
	//to achieve a baseline of what the runtime should be:
	double time[10];
	//we flush all of the sets -one by one - and then we invoke the victim to
run .
	double sum;
	for(i=0;i<100;i++){
	time[i] =measureOtherProcessRuntime(victimPid);
	sum+=time[i];
	}
	double averageRuntime=sum/100.0;
	for(i=0;i<100;i++){
	printf("i= %d . measureOtherProcessRuntime : %f\n",i,time[i]);
	}
	printf("Average runtime basline is : %f \n",averageRuntime);

	//The Evict and Time attack is very similar to prime and probe:
	//we evict all of the sets -one by one - and then we invoke the victim to
run
	//then we try to read the data again - and measure the time it took to
read.
	//(1) Evict the set   (2) let victim run (3) measure victim runtime.
	double runTimeEvictingSet[256];
	for(i=0;i<256;i++){
	int addressWeUsed= prime_Specific_Set(i); //(1) prime == evict
	//we reload from files to get the updated cache state and measure victim
runtime:
	runTimeEvictingSet[i]=measureOtherProcessRuntime(victimPid);
	}
	int wasSetUsed =0;
	for(i=0;i<256;i++){
	if (runTimeEvictingSet[i]>0.5)
		wasSetUsed =1;
	printf("i %d  runTimeEvictingSet = %f wasSetUsed = %d\n",i,
runTimeEvictingSet[i],wasSetUsed);
	wasSetUsed =0;
	}
	printf("******************Attacker MAIN ENDED SUCCESSFULLY
!**********************\n");
	free(globalMem);
```

```
	return 0;
}
```

Cache Collision

Victim_collision.c

```
#include "fileIOFunctions.h"
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <fcntl.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>
#define global_mem_size 65536
#define number_Of_Hex_Chars_For_Tag 5
#define number_Of_Hex_Chars_For_Set 2
#define number_Of_Hex_Chars_For_Offset 1
#define number_Of_Lines_In_Cache 256
#define micro_seconds_delay_for_cache_miss 500000 //half a second

//global arrays for the programmer level
unsigned char * globalMem ;
int pointerTounAllocatedMemory=0;
char** cacheTagArray;
char** cacheDataArray;

//Simple memory read/write methods for the programmer
unsigned char memRead(int Address){
	return readByteAtAddress(Address,globalMem,cacheTagArray,cacheDataArray);
}

void memWrite(unsigned char ByteToWrite,int Address){
	writeByteAtAddress(ByteToWrite,Address,
globalMem,cacheTagArray,cacheDataArray);
}

//signal function to invoke the victim to run:
void sigUsr2(int signo){ //code =12
	signal(SIGUSR2,sigUsr2);//reloading for backwords linux compatibillity
}


//We recieve the location of the plain ,sbox and key in memory (virtual
memory/DRAM)
void perform_AES_first_round(int offsetOfPlainInGlobalMem ,int
offsetOfSboxInGlobalMem,int OffsetOfKeyInGlobalMem){
```

```
	int i;
	unsigned char currKey,tableOutput,currPlain,xorRes;
	//creating pointers for easy access:
	unsigned char *plaintext=&globalMem[offsetOfPlainInGlobalMem];
	unsigned char *T0_Sbox= &globalMem[offsetOfSboxInGlobalMem];
	unsigned char *Key_arr= &globalMem[OffsetOfKeyInGlobalMem];
	printf("key array is :\n");
	for (i=0;i<16;i++){
	printf("%02X ",Key_arr[i]);
	}
	printf("\n");
	// now we perform the T0 table lookups for : Sbox [ key[i] xor plain[i]
]  as i is in {0 ,4,8,12}
	for(i=0;i<=12;i=i+4){
	currKey=memRead((int) &Key_arr[i]);
	currPlain=memRead((int) &plaintext[i]);
	xorRes=(currKey^currPlain);
	//Now we perform the table lookup:
	tableOutput=memRead(&T0_Sbox[xorRes]);
	printf("i=%d : curr key = %02X ,currPlain =%02X ,xorRes
=%02X,tableOutput= %02X \n",i,currKey,currPlain,xorRes,tableOutput);
	}

}
```

```
void put_256_byte_Sbox_In_Global_Mem (int OffsetInGlobalMem){
	//(This is similar to Rijndael S-box  -hard coded .(not identical ,but
using the same values .see: https://en.wikipedia.org/wiki/Rijndael_S-box)
	unsigned char Sbox [256]=
                        {   0x63 ,0x7C ,0x77 ,0x7B ,0xF2 ,0x6B ,0x6F ,0xC5
,0x30 ,0x01 ,0x67 ,0x2B ,0xFE ,0xD7 ,0xAB ,0x76 ,
                        0x10 ,0xCA ,0x82 ,0xC9 ,0x7D ,0xFA ,0x59 ,0x47 ,0xF0
,0xAD ,0xD4 ,0xA2 ,0xAF ,0x9C ,0xA4 ,0x72 ,
                        0xC0 ,0x20 ,0xB7 ,0xFD ,0x93 ,0x26 ,0x36 ,0x3F ,0xF7
,0xCC ,0x34 ,0xA5 ,0xE5 ,0xF1 ,0x71 ,0xD8 ,
                        0x31 ,0x15 ,0x30 ,0x04 ,0xC7 ,0x23 ,0xC3 ,0x18 ,0x96
,0x05 ,0x9A ,0x07 ,0x12 ,0x80 ,0xE2 ,0xEB ,
                        0x27 ,0xB2 ,0x75 ,0x40 ,0x09 ,0x83 ,0x2C ,0x1A ,0x1B
,0x6E ,0x5A ,0xA0 ,0x52 ,0x3B ,0xD6 ,0xB3 ,
                        0x29 ,0xE3 ,0x2F ,0x84 ,0x50 ,0x53 ,0xD1 ,0x00 ,0xED
,0x20 ,0xFC ,0xB1 ,0x5B ,0x6A ,0xCB ,0xBE ,
                        0x39 ,0x4A ,0x4C ,0x58 ,0xCF ,0x60 ,0xD0 ,0xEF ,0xAA
,0xFB ,0x43 ,0x4D ,0x33 ,0x85 ,0x45 ,0xF9 ,
                        0x02 ,0x7F ,0x50 ,0x3C ,0x9F ,0xA8 ,0x70 ,0x51 ,0xA3
,0x40 ,0x8F ,0x92 ,0x9D ,0x38 ,0xF5 ,0xBC ,
                        0xB6 ,0xDA ,0x21 ,0x10 ,0xFF ,0xF3 ,0xD2 ,0x80 ,0xCD
,0x0C ,0x13 ,0xEC ,0x5F ,0x97 ,0x44 ,0x17 ,
                        0xC4 ,0xA7 ,0x7E ,0x3D ,0x64 ,0x5D ,0x19 ,0x73 ,0x90
,0x60 ,0x81 ,0x4F ,0xDC ,0x22 ,0x2A ,0x90 ,
                        0x88 ,0x46 ,0xEE ,0xB8 ,0x14 ,0xDE ,0x5E ,0x0B ,0xDB
,0xA0 ,0xE0 ,0x32 ,0x3A ,0x0A ,0x49 ,0x06 ,
                        0x24 ,0x5C ,0xC2 ,0xD3 ,0xAC ,0x62 ,0x91 ,0x95 ,0xE4
,0x79 ,0xB0 ,0xE7 ,0xC8 ,0x37 ,0x6D ,0x8D ,
                        0xD5 ,0x4E ,0xA9 ,0x6C ,0x56 ,0xF4 ,0xEA ,0x65 ,0x7A
,0xAE ,0x08 ,0xC0 ,0xBA ,0x78 ,0x25 ,0x2E ,
                        0x1C ,0xA6 ,0xB4 ,0xC6 ,0xE8 ,0xDD ,0x74 ,0x1F ,0x4B
,0xBD ,0x8B ,0x8A ,0xD0 ,0x70 ,0x3E ,0xB5 ,
```

```
                        0x66 ,0x48 ,0x03 ,0xF6 ,0x0E ,0x61 ,0x35 ,0x57 ,0xB9
,0x86 ,0xC1 ,0x1D ,0x9E ,0xE0 ,0xE1 ,0xF8 ,
                        0x98 ,0x11 ,0x69 ,0xD9 ,0x8E ,0x94 ,0x9B ,0x1E ,0x87
,0xE9 ,0xCE ,0x55 ,0x28 ,0xDF ,0xF0 ,0x8C
                        };
      int i;
      int runningOffset=OffsetInGlobalMem;
      //copy Sbox array into global mem
      for (i=0;i<256;i++){
      globalMem[runningOffset]=Sbox[i];
      runningOffset++;
      }

pointerTounAllocatedMemory=pointerTounAllocatedMemory+(16*16);//allocating 16
blocks to the 256 sbox

      }

      void TesterFor_put_256_byte_Sbox_In_Global_Mem(){
      int i=0;
      //Test for 256 byte Sbox :
      int sbox_offsetInGlobalMem=pointerTounAllocatedMemory;
      put_256_byte_Sbox_In_Global_Mem(sbox_offsetInGlobalMem);
      unsigned char sboxReadFromMemory[256];
      for (i=0;i<256;i++){

sboxReadFromMemory[i]=memRead((int)&globalMem[sbox_offsetInGlobalMem+i]);
      }
      //Reading and printing back the  values we read:
      printf("\n **********The sbox is : *********\n");
      for (i=0;i<256;i++){
            printf("%02X ,",sboxReadFromMemory[i]);
            if(i%15==0)
            printf("\n");
      }

      }



void put_128_bit_key_into_global_mem (int offsetOfKeyInGlobalMem){
      //Extension : using 1 set for the key : loading 16 bytes into memory
      unsigned char key_16_bytes
[16]={0x55,0x43,0xAF,0x68,0x91,0xDB,0xCC,0xE5,0x78,0xD3,0xA9,0x18,0x99,0xCE,0x6
D,0xB8};
      int i;
      int Running_offsetOfKeyInGlobalMem=offsetOfKeyInGlobalMem;
      pointerTounAllocatedMemory+=16;//allocating space for it
      //copying into global mem (DRAM)
      for(i=0;i<16;i++){
      globalMem[Running_offsetOfKeyInGlobalMem]=key_16_bytes[i];
      Running_offsetOfKeyInGlobalMem++;
      }
}



//Receives a pointer to a 16 bit array which we will fill with data read from
file:
//This method assumes
void readPlaintextFromFile (unsigned char * plaintext){
      FILE *fptr;
```

```
	fptr = fopen("plainTextToEncrpt.txt","r");
	int i;
	const unsigned MAX_LENGTH = 82;
	char buffer[MAX_LENGTH];
	//reading the line from the file into a buffer.
	fgets(buffer, MAX_LENGTH, fptr);
	//reading 16 values from buffer and converting to unsigned chars (bytes)
	//code is similar to "getByteArrayOfACacheDataLine()".
	char byteBuffer[5];
	byteBuffer[4]='\0';
	int indexOfStartOfString=0;
	for (i=0;i<16;i++){
	byteBuffer[0]=buffer[indexOfStartOfString];
	byteBuffer[1]=buffer[indexOfStartOfString+1];
	byteBuffer[2]=buffer[indexOfStartOfString+2];
	byteBuffer[3]=buffer[indexOfStartOfString+3];
	//converting "0xHH" to a byte:
	int numericalValue=(int)strtol(byteBuffer,NULL,0);
	plaintext[i]=numericalValue;
	indexOfStartOfString+=5;
	}
	fclose(fptr);
}

//method that reads from file then writes the data to the global memory.
Returns the offset in global memory which it is stored at.
int getPlainText_AndPutInGlobalMemory(){
	int i;
	unsigned char plaintextReadFromFile[16];
	readPlaintextFromFile(plaintextReadFromFile);
	//put plainText into global memory:
	int offsetOfPlainTextInGlobalMem=pointerTounAllocatedMemory;
	pointerTounAllocatedMemory+=16;
	for(i=0;i<16;i++){
	globalMem[i+offsetOfPlainTextInGlobalMem]=plaintextReadFromFile[i];
	}
	return offsetOfPlainTextInGlobalMem;
}




//we let control of the cpu to the other process
// before we do that we need to update the cache files with the current state.
void signalOtherProccessToRun(int pid ){
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	kill(pid,SIGUSR2);
}
void printPlainText (int offsetOfPlainTextInGlobalMem){
	int i;
	printf("plaintext is : ");
	for(i=0;i<16;i++){
	printf("%02X,",globalMem[i+offsetOfPlainTextInGlobalMem]);
	}
	printf("\n");
}
```

```
int main(){
	int i;
	writeVictimPidToFile();
	//loading the signal sigusr2
	signal(SIGUSR2,sigUsr2);
	//defining the global memory space (in DRAM) for the process:
	globalMem=(unsigned char *)malloc(global_mem_size*(sizeof(unsigned
char)));
	cacheTagArray = loadCacheTagFromFile();
	cacheDataArray=loadCacheDataArrayFromFile();
	//put_256_byte_Sbox_In_Global_Mem(sbox_offsetInGlobalMem);
	int sbox_offsetInGlobalMem=pointerTounAllocatedMemory;
	put_256_byte_Sbox_In_Global_Mem(sbox_offsetInGlobalMem);
	//put 16 byte key onto global memory:
	int offsetOfKeyInglobalMem=pointerTounAllocatedMemory;
	put_128_bit_key_into_global_mem(offsetOfKeyInglobalMem);
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);

	//Infinite loop : waiting for plaintext to encrypt and executing AES
first round on the plainText
	int counter=0;
   while(1){
	pause();//wainting for sigusr2 signal from the system (the attacker can
invoke it to run)
	int callingProcessPid =readAttackerPidFromFile();
	counter++;
	printf("Victim is invoked and running for the %d time  :\n",counter);
	printf("Performing first round of AES :\n");
	cacheTagArray = loadCacheTagFromFile();
	cacheDataArray=loadCacheDataArrayFromFile();
	int offsetOfPlainTextInGlobalMem=getPlainText_AndPutInGlobalMemory();
	printPlainText(offsetOfPlainTextInGlobalMem);
	//Do_Some_Cryptography:

perform_AES_first_round(offsetOfPlainTextInGlobalMem,sbox_offsetInGlobalMem,off
setOfKeyInglobalMem);
	//Returning control to the calling process:
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	signalOtherProccessToRun(callingProcessPid);
	printf("******victim ended -waiting for another invocation******\n");
	//reallocating space for plaintext:
	pointerTounAllocatedMemory-=16;
	}
	free (globalMem);
	return 0;
}
```

---

attacker_collison.c

```
#include "fileIOFunctions.h"
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <fcntl.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>
#define global_mem_size 65536
#define number_Of_Hex_Chars_For_Tag 5
#define number_Of_Hex_Chars_For_Set 2
#define number_Of_Hex_Chars_For_Offset 1
#define number_Of_Lines_In_Cache 256
#define micro_seconds_delay_for_cache_miss 500000 //half a second

//global arrays for the programmer level
unsigned char * globalMem ;
int pointerTounAllocatedMemory=0;
char** cacheTagArray;
char** cacheDataArray;

//Simple memory read/write methods for the programmer
unsigned char memRead(int Address){
      return readByteAtAddress(Address,globalMem,cacheTagArray,cacheDataArray);
}

void memWrite(unsigned char ByteToWrite,int Address){
      writeByteAtAddress(ByteToWrite,Address,
globalMem,cacheTagArray,cacheDataArray);
}

//signal function that the victim process can use to ell us that it has
finished processing/waiting and it's our turn to run
void sigUsr2(int signo){ //code =12
      signal(SIGUSR2,sigUsr2);//reloading for backwards linux compatibility
}



//measure Runtime of the victim process -by signaling it and waiting for it to
signal back to us .
double measureOtherProcessRuntime(int victimPid){
```

```
	//setup:
	struct timeval  startTime, endTime;
	gettimeofday(&startTime, NULL);
	//5 lines for process switching:
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	signalOtherProccessToRun(victimPid); //letting the victim run
	pause(); //waiting for victim to finish
	cacheTagArray = loadCacheTagFromFile();
	cacheDataArray=loadCacheDataArrayFromFile();
	gettimeofday(&endTime, NULL);
	double time=(double) (endTime.tv_usec - startTime.tv_usec) / 1000000 +
(double) (endTime.tv_sec - startTime.tv_sec);
	return time;

}


//we let control of the cpu to the other process
// before we do that we need to update the cache files with the current state.
void signalOtherProccessToRun(int pid ){
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	kill(pid,SIGUSR2);
}

//plaintext is 128 bits long = 16 bytes .we write it to a file so the victim
process can read and encrypt it.
void writePlainIntoFile(unsigned char * plainTextByteArray){
	FILE *fptr;
	fptr = fopen("plainTextToEncrpt.txt","w");
	char StringTowriteTofile[150]="";
	char buffer[10]="";
	int i;
	//creating the string to write to file
	for(i=0;i<16;i++){
	sprintf(buffer,"0x%02X ",plainTextByteArray[i]);
	strcat(StringTowriteTofile,buffer);
	}
	fprintf(fptr,"%s",StringTowriteTofile);
	fclose(fptr);
}

//We will change the plaintext in order to test for a specific xor result (x)
between the upper halves of plain[i] and plain[j]
void changePlaintext(int i,int j,int x,unsigned char ValueForPadding,unsigned
char *plain_16_bytes){
	int index;
	//writing the ValueForPadding to plain
	for(index=0;index<16;index++){
	plain_16_bytes[index]=ValueForPadding;
	}
	unsigned char plain_j = 0;
	unsigned char plain_i = x*16; //we write X into the upper half of p_i
(shift left 4 times)
	//now <p_i> xor <p_j> = (xor between upper 4 bits) =  x xor 0 = x.
	plain_16_bytes[i]=plain_i;
	plain_16_bytes[j]=plain_j;
	//Now <plain[i]> xor <plain[j]> = x .
}

//Flushing a range of sets from the cache :
void flushSboxLinesFromCache(int setSboxStartsAt , int setSboxEndsAt ){
	int i=setSboxStartsAt;
	for(i=setSboxStartsAt;i<=setSboxEndsAt;i++){
```

```
	flush(i,cacheTagArray,cacheDataArray);
	}
}

//we want to print the xors we learned :
//K[0] xor K[4] , , K[0] xor K[8], K[0] xor K[12],K[4] xor K[8],K[4] xor
K[12],K[8] xor K[12],
void printAttackResults(double *** time){
	int i,j,xorRes, indexOfMin=0;
	double minTime =2147483647;
	for(i=0;i<=12;i++){
	for(j=i+4;j<=12;j++){
		//now we need to find the minimum index out of 0-15
		minTime =2147483647;
		indexOfMin=0;
		for(xorRes=0;xorRes<16;xorRes++){
		if( time[i][j][xorRes] <   minTime){ //new minimum
			minTime=time[i][j][xorRes];
			indexOfMin=xorRes;
		}
		printf ( "<K_%d> xor <K_%d> = %d  \n",i,j,indexOfMin);
		}
	}
	}
}


int main(){
	//loading the signal sigusr2
	signal(SIGUSR2,sigUsr2);
	cacheTagArray = loadCacheTagFromFile();
	cacheDataArray=loadCacheDataArrayFromFile();
	//defining the global memory space (in DRAM) for this process:
	globalMem=(unsigned char *)malloc(global_mem_size*(sizeof(unsigned
char)));
	writeAttackerPidToFile();
	int victimPid =readVictimPidFromFile();
	double excutionTimeAfterEvicting[256];
	flushSboxLinesFromCache(0,255);
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	//return 0;

	unsigned char plain_16_bytes
[16]={0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0
0,0x00};
	//writePlainIntoFile(plain_16_bytes);
	//Creating the Time (i , j , xorRes) array to measure runtime for all
combinations of key bytes and xor results.
	double time[16][16][16];
	int i,j,xorRes ,paddingIndex=0;
	unsigned char valuesForPadding[4]={0,0xFF,0xAA,0x77};
	//To shorten the runtime we will only calculate pairs in {0,4,8,12}
	for(i=0;i<16;i=i+4){
	for(j=0;j<16;j=j+4){
		if(j==i){
		continue;//we don't need to measure time for <pi> xor <pi> because
it doesnt give as information
		}
		for (xorRes=0;xorRes<16;xorRes++){ //xorRes will iterate over the
options for xor res between plain[i] and plain[j]
		for(paddingIndex=0;paddingIndex<4;paddingIndex++){
			//flushing the cache to ensure that we don't get collisions
between different runs.
```

```
                  flushSboxLinesFromCache(0,50);
                  //choose a plaintext to test i,j,xorRes:

changePlaintext(i,j,xorRes,valuesForPadding[paddingIndex],plain_16_bytes);
                  //writing plain to a file before invoking the victim to run
and encrypt the plaintext.
                  writePlainIntoFile(plain_16_bytes);
                  //running and timing the victim with our changed plaintext:
                  //Dividing by 4 -to get an average out of 4 measurements
                  time[i][j][xorRes]+=measureOtherProcessRuntime(victimPid)/4;

            }

            }
      }

      }
      //Attack ended - now we can print the time measurements for each i and j
pair :
      printf(" ****************TIME RESULTS FOR THE CACHE COLLISION
**********\n");
      //print the results :
      for(i=0;i<16;i=i+4){
      for(j=0;j<16;j=j+4){
            if(j==i)
            continue;//we don't need to measure time for <pi> xor <pi> because
it doesn't give as information

            printf("\nTime for i= %d j= %d is : \n",i,j);
            for (xorRes=0;xorRes<16;xorRes++){
                  printf(" %f ",time[i][j][xorRes]);
            }
      }
      }
      //printAttackResults - printing what we have learned about the secret key
(upper half of xor result between key's bytes).
      printf("\n The results for the cache collision attack -what we know about
the key :\n");
      int  indexOfMin=0;
      double minTime =2147483647;
      for(i=0;i<=12;i=i+4){
      for(j=i+4;j<=12;j=j+4){
            //now we need to find the minimum index out of 0-15
            minTime =2147483647;
            indexOfMin=0;
            for(xorRes=0;xorRes<16;xorRes++){
             if( time[i][j][xorRes] <   minTime){ //new minimum
                  minTime=time[i][j][xorRes];
                  indexOfMin=xorRes;
            }

            }
            printf("\n");
            printf ( "<K_%d> xor <K_%d> = %d  \n",i,j,indexOfMin);
      }
      }
      printf("*****************Attacker MAIN ENDED SUCCESSFULLY
!*********************\n");
      free(globalMem);
      return 0;
}
```

fileIoFunctions.h -simulation functions for all files :

```
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <fcntl.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>

#define number_Of_Hex_Chars_For_Tag 5
#define number_Of_Hex_Chars_For_Set 2
#define number_Of_Hex_Chars_For_Offset 1
#define number_Of_Lines_In_Cache 256
#define micro_seconds_delay_for_cache_miss 500000 //half a second?


void free2dArray(char ** arr,int rows);
///
///*************************************************************
//Data array METHODS
//*************************************************************
char ** dymaciacllyAlocateCharArr (int lines,int columns){
      char ** Arr=(char **)malloc(sizeof(char*)*lines);
      int i=0;
      for (i=0;i<lines;i++){
      Arr[i]=(char*)malloc(sizeof(char)*columns);
      }
      return Arr;
}
//
char ** loadCacheDataArrayFromFile(){
      int i,j;
      char** cacheDataArray
=dymaciacllyAlocateCharArr(number_Of_Lines_In_Cache,81);

      FILE *fptr;
      fptr = fopen("cacheDataArray.txt","r");
      const unsigned MAX_LENGTH = 82;
      char buffer[MAX_LENGTH];
      i=0;

      //reading from the file line by line
      while (fgets(buffer, MAX_LENGTH, fptr)){
      strcpy(cacheDataArray[i],buffer);
      i++;
      }
   fclose(fptr);
   return cacheDataArray;
}
//writing the array to the file when the process finishes running or
void writeToDataArraryFile(char ** cacheDataArray){
   int i;
   FILE *fptr;
      fptr = fopen("cacheDataArray.txt","w");
```

```
	for(i=0;i<number_Of_Lines_In_Cache;i++){
	fprintf(fptr,"%s",cacheDataArray[i]);
	}

	fclose(fptr);

}
//retruns a byte array (char array) each char is a byte->we return 16 bytes
array which is a cache line/block
unsigned char* getByteArrayOfACacheDataLine(int set ,  char **cacheDataArray ){
	int i;
	unsigned char* byteLineArr=(unsigned char*) malloc(sizeof(unsigned
char)*16);
	char *lineString = cacheDataArray[set];
	char buffer[5];
	buffer[4]='\0';
	int indexOfStartOfString=0;
	for(i=0;i<16;i++){
	  buffer[0]=lineString[indexOfStartOfString];
	buffer[1]=lineString[indexOfStartOfString+1];
	buffer[2]=lineString[indexOfStartOfString+2];
	buffer[3]=lineString[indexOfStartOfString+3];
	int numericalValue=(int)strtol(buffer,NULL,0);
	unsigned char byteValue=(char)   numericalValue%256;
	byteLineArr[i]=byteValue;
	indexOfStartOfString+=5;

	}
	return byteLineArr;
}

//Important : if we initialize manually the byteArray the length should be
const .
//meaning char arr[16]={0x1,0xFF...}. NOT char * arr !!
void writeByteArrayToACacheDataLine(int setToUpdate ,unsigned char* byteArray ,
char **cacheDataArray ){
	//we will build this line and then  copy this line to the cacheDataArray:
	char wholeLine[82]="";
	//template for string:
	char number_buffer[6]={'0','x','0','0','0','\0'};
	int i=0 ,index_at_string=0;
	for(i=0;i<16;i++){
	number_buffer[2]='0';
	int startIndex=2;
	if(byteArray[i]<16){
		startIndex=3;
	}
	sprintf(&number_buffer[startIndex],"%X ",byteArray[i]);
	//printf("buffer is :%s.\n",number_buffer);
	strcat(wholeLine,number_buffer);
	}
   // printf("\nwriteByteArrayToACacheDataL:whole line is : %s\n",wholeLine);
	strcat(wholeLine,"\n");

	strcpy(cacheDataArray[setToUpdate],wholeLine);

}

//use this method to get a specific block at a cache Line at a known set.
unsigned char getSpecficByteFromDataArray(int set,int offsetInBlock, char
**cacheDataArray){
	unsigned char * dataLineAtSet
=getByteArrayOfACacheDataLine(set,cacheDataArray);
	int i=0;
	unsigned char byte =dataLineAtSet[offsetInBlock];
	free(dataLineAtSet);
	return byte;
}
//must check before hand that the tag is equal !  only then we can change bytes
in the cache data array at a specific offset.
```

```
void writeByteToDataArray(unsigned char byteTowrite,int set,int
offsetInBlock,char **cacheDataArray){
	unsigned char * dataLineAtSet
=getByteArrayOfACacheDataLine(set,cacheDataArray);
	dataLineAtSet[offsetInBlock]=byteTowrite;
	writeByteArrayToACacheDataLine(set,dataLineAtSet,cacheDataArray);
	free(dataLineAtSet);
}

void testerForDataArray(){
	int i;
	printf("********************\nMain is running\n");
	char **cacheDataArray=loadCacheDataArrayFromFile();

	cacheDataArray[1][2]='D';
	cacheDataArray[1][3]='5';
	//cacheDataArray[1][0]='1';
	writeToDataArraryFile(cacheDataArray);

	unsigned char* DataLine_At_Set_0
=getByteArrayOfACacheDataLine(0,cacheDataArray);
	unsigned char byte = getSpecficByteFromDataArray(1,0,cacheDataArray);
	printf("byte = %X\n",byte);
	byte = getSpecficByteFromDataArray(2,2,cacheDataArray);
	printf("2,2  = %X\n",byte);
	printf("DataLine_At_Set_0 is :\n");
	for(i=0;i<16;i++){
	printf("%d=%X ,",DataLine_At_Set_0[i],DataLine_At_Set_0[i]);
	}


	unsigned char
byteArrayTowrite[16]={0x55,0xFF,0x6d,0x55,0xFF,0x6d,0x55,0xFF,0x6d,0x55,0xFF,0x
6d,0x6d,0x6d,0x6d,0x6d};
	writeByteArrayToACacheDataLine(2,byteArrayTowrite,cacheDataArray);

	writeByteToDataArray(0x30,3,0,cacheDataArray);
	writeByteToDataArray(0xFF,3,15,cacheDataArray);
	writeByteToDataArray(0xFF,3,7,cacheDataArray);
	writeByteToDataArray(0xFF,0,14,cacheDataArray);
	writeToDataArraryFile(cacheDataArray);
	////freeing arrays
	free2dArray(cacheDataArray,number_Of_Lines_In_Cache);
	free(DataLine_At_Set_0);
	printf("End Of Main ********************\n");
}


//*****************************************************************
//TAG METHODS
//*****************************************************************
//helper method :
//works for 32 bit integers - returns 1 or 0 - the value of the requested bit.
int getSpecificBitFromInteger(int integer ,int indexOfBit){
	return (integer >> indexOfBit) & 1;
}

int convertStringTagToNumber (char * tagString){
	return strtol(tagString,NULL,0);
}

void print2dArray(char ** arr,int rows,int columns){
	int i,j;
	for (i=0;i<rows;i++){
	printf("%s\n",arr[i]);
	}
}
//set is the index of the line in the cache
char * getTagAtSet (int set,char ** cacheTagArr){
	return cacheTagArr[set];
}
```

```
void changeTagAtSet(int set,char * newTagString,char ** cacheTagArr){
	if(strlen(newTagString)>9){
	printf("error at -changeTagAtSet- input string too long(>9)");
	}
	cacheTagArr[set]=strcpy(cacheTagArr[set],newTagString);
}

//This function reads the Txt file that has the simulated Cache tag array - and
stores it in a byte array of 256 *9  which it then returns
//(256 lines - 9 characters)
char ** loadCacheTagFromFile(){
	//first we load the tag arr
	char** cacheTagArray=
dymaciacllyAlocateCharArr(number_Of_Lines_In_Cache,8);
	int i=0,j=0;
	FILE *fptr;
	char buffer[10]="";
	fptr = fopen("cacheTagArray.txt","r");
	//going over the whole file and copying the 256 tag values into the array
;
	for(i=0;i<number_Of_Lines_In_Cache;i++){
	fscanf(fptr,"%s",buffer);
	for(j=0;j<8;j++){
		cacheTagArray[i][j]=buffer[j];
	}
	}

	fclose(fptr);
	return cacheTagArray;

}
//Overwrites the cacheTagArray.txt file with an updated array from our program.
//SHOULD BE CALLED WHENEVER WE STOP THE EXECUTION/WAIT/SLEEP ETC- BECAUSE IT
UPDATES THE CACHE THAT THE OTHER PROCESS SEES!
void writeToTagArrayFile(char **cacheTagArray){
	FILE * fptr;
	fptr= fopen("cacheTagArray.txt","w");
	int i;
	for(i=0;i<number_Of_Lines_In_Cache;i++){
	fprintf(fptr,"%s\n",cacheTagArray[i]);
	}
	fclose(fptr);
}

//Tester:
void testerForTagFile(){
	struct timeval  startTime, endTime;
	gettimeofday(&startTime, NULL);
	char **cacheTagArray;;
	cacheTagArray=loadCacheTagFromFile();
	print2dArray(cacheTagArray,number_Of_Lines_In_Cache,8);
	char * tag1str =getTagAtSet(0,cacheTagArray);
	printf("string of tag at set 0 is :%s \n",tag1str);
	printf("numerical value is : %d\n",strtol(tag1str,NULL,0));
	char *str ="0xFFFFF";
	changeTagAtSet(2,str,cacheTagArray);
	changeTagAtSet(1,"0x54ADC",cacheTagArray);
	writeToTagArrayFile(cacheTagArray);

	gettimeofday(&endTime, NULL);
	printf ("Total time = %f seconds\n",
	(double) (endTime.tv_usec - startTime.tv_usec) / 1000000 +
	(double) (endTime.tv_sec - startTime.tv_sec));
}
//****************************************************************
//Helper method.
void free2dArray(char ** arr,int rows){
	int i;
	for (i=0;i<rows;i++){
	free(arr[i]);
```

```
	}
	free(arr);
}
////////////////////////////////////////////////////////////////////////
//Exchanging Pid's
////////////////////////////////////////////////////////////////////
void writeAttackerPidToFile(){
	int pid =getpid();
	FILE *fptr;
	fptr = fopen("Attacker_PID.txt","w");
	char buffer[100]="";
	sprintf(buffer,"%d",pid);
	fprintf(fptr,"%s",buffer);
	fclose(fptr);
}
int readVictimPidFromFile(){
	FILE *fptr;
	fptr = fopen("Victim_PID.txt","r");
	char buffer[100]="";
	fscanf(fptr,"%s",buffer);
	int pid =atoi(buffer);
	printf("the pid of the victim is %d\n",pid);
	fclose(fptr);
	return pid;
}

void writeVictimPidToFile(){
	int pid =getpid();
	FILE *fptr;
	fptr = fopen("Victim_PID.txt","w");
	char buffer[100]="";
	sprintf(buffer,"%d",pid);
	fprintf(fptr,"%s",buffer);
	fclose(fptr);
}
int readAttackerPidFromFile(){
	FILE *fptr;
	fptr = fopen("Attacker_PID.txt","r");
	char buffer[100]="";
	fscanf(fptr,"%s",buffer);
	int pid =atoi(buffer);
	printf("the pid of the attacker is %d\n",pid);
	fclose(fptr);
	return pid;
}

///*****************************************************************
//CPU PART
//*****************************************************************
//helper method :
int convertStringToNumber (char * String){
	// strtol(tagString,NULL,0) takes a string in format 0xYYYYY.. and
converts to int.
	printf("\nstr  we get in convertStringToNumber: %s \n",String);
	int x=(int)strtol(String,NULL,0);
	return x;
}
//helper method ://dynamic! format of output 0xYYYYY
char * convertTagNumberToTagstring(int numericalTag){
	char * tagStr =(char*) malloc (sizeof(char)*8);
	tagStr[0]='0';
	tagStr[1]='x';
	//we add the text from the third character
	sprintf(&tagStr[2], "%05X", numericalTag);
	return tagStr;
}


//Address Decoding:
int getAddress_Set (int address){
	int set = address/16;
```

```
	set = set%256;
	return set;
}

int getAddress_Offset (int address){
	int offset = address%16;
	return offset;
}

int getAddress_Tag (int address){
	int tag = address/4096;
	return tag;
}
//returns 1 if address is in a block that is stored in cache .0 otherwise.
int checkIfAddressIsIncache(int Address,char** cacheTagArray){
	int mySet, myTag, boolean;
	mySet = getAddress_Set(Address); //numerical
	myTag = getAddress_Tag(Address);//numerical
	char tag[8] = "0x12345";//initializing the arrays.
	char str1[8] = "0x";
	sprintf(tag, "%05X", myTag);
	strcat(str1,tag);//creating the formatted string 0xYYYYY
	boolean = strcmp(str1, cacheTagArray[mySet]);
	if (boolean == 0){
	return 1;
	}
	else{
	return 0;
	}
}

//writing both arrays to the cache - should be called every time we "leave" the
cpu or let another process to run (waiting)
void UpdateCacheTextFiles(char** cacheTagArray,char ** cacheDataArray){
	writeToDataArraryFile(cacheDataArray);
	writeToTagArrayFile(cacheTagArray);

}
void freeBothArrays(char** cacheTagArray,char ** cacheDataArray){
	free2dArray(cacheTagArray,number_Of_Lines_In_Cache);
	free2dArray(cacheDataArray,number_Of_Lines_In_Cache);
}



//Aritificial time delay for cache miss .
void waitForCacheMissDelay(){
	usleep(micro_seconds_delay_for_cache_miss);
}

//writes zeroes to the tag and the array.
void flush(int setToFlush,char** cacheTagArray,char** cacheDataArray){
	//use changeTagAtSet () , and writeByteArrayToACacheDataLine -
defineunsigned char *=  {0,0..16 times}
	if(setToFlush>number_Of_Lines_In_Cache ){
	printf("Error in flush method - set %d outside of range of
sets",setToFlush);
	return;
	}
	//changing the tag and the data array to be all zeros.
	changeTagAtSet (setToFlush,"0x00000",cacheTagArray);
	unsigned char
zerosByteArray[16]={0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
,0x00,0x00,0x00,0x00};
	writeByteArrayToACacheDataLine(setToFlush,zerosByteArray,cacheDataArray);
}

	//This method will be called when the cpu decides to evict a cache line
at a specific set and replace it with a new line (new tag and data)
	//Important : if we initialize manually the byteArray the length should
be const .
```

```
	//meaning char arr[16]={0x1,0xFF...}. NOT char * arr !!
	void evictSet (int set,char *newTag ,unsigned char*
newDataByteArray,char** cacheTagArray,char** cacheDataArray)
	{
	//changing the tag and data at the line:
	changeTagAtSet (set,newTag,cacheTagArray);
	writeByteArrayToACacheDataLine(set,newDataByteArray,cacheDataArray);
	}


	//preforming the memory reads.We need access to the global memory so we
can
	unsigned char  readByteAtAddress(int address,unsigned char
*globalMem,char** cacheTagArray,char** cacheDataArray ){
	//firstly we decode/breaak the address into its tag ,set and offset:
	int set=getAddress_Set(address);
	int tag=getAddress_Tag(address);
	int offset=getAddress_Offset(address);
	//Then -we want to check if the address/block the address is in - is
already in cache :
	int
isAddressAlreadyIncache=checkIfAddressIsIncache(address,cacheTagArray);
	if(isAddressAlreadyIncache==0)
	{
		////The address is not currently in cache-> we should evict and
replace with new tag data!
		//also - add a artificial/simulated cache miss delay of (1/2
second?)
		waitForCacheMissDelay();
		//now we need to read 16 bytes from global memory and then copy
that line to the cache:
		//assumption - the address is either aligned to a start of block -
or it is possible to go back and read bytes
		//that are "behind" it . (i.e. if offset=2 we need to able to
accsess (address -1) (address -2) etc.)
		printf("readByteAtAddress: cache miss:evicting set %d ,new tag is
:0x%05X \n",set,tag);
		unsigned char byteArr
[16]={0x00,0x00,0x00,0x00,0x00,0x0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
,0x00};
		int i;

		int AddressTheBlockStartsAt=address-offset;
		unsigned char * memoryPointer =(unsigned char
*)AddressTheBlockStartsAt;

		//copying from Dram(global memory) to the byteArr
		for (i=0;i<16;i++){
		byteArr[i]=memoryPointer[i];
		// printf("now byteArr[%d]=%d\n",i,byteArr[i]);
		}
		//now we create the new tag string;
		char *tagStr=convertTagNumberToTagstring(tag);//dynamic -should be
freed
		// eviting the set - with new tag and data :
		evictSet(set,tagStr,byteArr,cacheTagArray,cacheDataArray);
		free(tagStr);
	}
	else{
		printf("readByteAtAddress:CACHE HIT ON ADDRESS %d SET %d and Tag
=0x%05X\n",address,set,tag);
	}
	//Now ,we know that the data is in cache . We return the byte that is
stored in the cache:
	unsigned char byte
=getSpecficByteFromDataArray(set,offset,cacheDataArray);
	return byte;
	}

	// We have a write-through policy.If the address isn't in cache its'
block will be loaded .
```

```
	//when we write to memory we update both cache and global Memory(DRAM).
	void writeByteAtAddress(unsigned char byteTowrite,int address,unsigned
char *globalMem,char** cacheTagArray,char** cacheDataArray){
	//Decoding addresses:
	int i=0;
	int set=getAddress_Set(address);
	int tag=getAddress_Tag(address);
	int offset=getAddress_Offset(address);
	int AddressTheBlockStartsAt=address-offset;
	unsigned char * memoryPointer =(unsigned char *)AddressTheBlockStartsAt;
	//Then -we want to check if the line/block the address is in - is already
in cache :
	int
isAddressAlreadyIncache=checkIfAddressIsIncache(address,cacheTagArray);
	if(isAddressAlreadyIncache==0){
		////The address is not currently in cache-> we should evict and
replace with new tag data!
		//also - add a artificial/simulated cache miss delay of (1/2
second?)
		waitForCacheMissDelay();
		unsigned char byteArr
[16]={0x00,0x00,0x00,0x00,0x00,0x0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
,0x00};

		//copying from Dram(global memory) to the byteArr
		for (i=0;i<16;i++){
		byteArr[i]=memoryPointer[i];
		}
		//now we create the new tag string;
		char *tagStr=convertTagNumberToTagstring(tag);//dynamic -should be
freed
		// :eviting the set - with new tag and data :
		printf("writeByteAtAddress:cache miss on address %d now we
evict:set=%d,tagStr=%s\n",address,set,tagStr);
		evictSet(set,tagStr,byteArr,cacheTagArray,cacheDataArray);
		free(tagStr);

	}
	else{
		printf("writeByteAtAddress:CACHE HIT ON ADDRESS %d SET %d and Tag
=0x%05X\n",address,set,tag);
	}
	//now we write the Byte to the cache and global Memory :
	memoryPointer[offset]=byteTowrite; //write to global Memory
	writeByteToDataArray(byteTowrite,set,offset,cacheDataArray);//write to
cache
	}

//Helper method initializes the sbox onto the DRAM - initial state of the
program.
void putSboxOntoTheGlobalMem(int offsetInsideGlobalMem,unsigned char
*globalMem){
	int i;
	//mapping of 8 bits to 8 bits.sbox is precalculated:
	//sbox is hard coded
	unsigned char
sbox[16]={0x0F,0x03,0x07,0x0A,0x0E,0x04,0x0D,0x01,0x00,0x05,0x6,0x0C,0x2,0x09,0
xB,0x08};
	for(i=0;i<16;i++){
	globalMem[offsetInsideGlobalMem+i]=sbox[i];
	}

}


void testerFor_flush_evictSet_waitForCacheMissDelay(){
	int mySet1 = 1269764;
	char ** cacheTagArray;
	cacheTagArray = loadCacheTagFromFile();
	char ** cacheDataArray=loadCacheDataArrayFromFile();
	flush(4,cacheTagArray,cacheDataArray);
```

```
	unsigned char
byteArrayForSet3[16]={0x13,0x23,0x33,0x43,0x03,0x50,0x06,0x07,0x1d,0x05,0x2,0x0
0,0x2,0x10,0x9,0x1};
	evictSet(4,"0x12345",byteArrayForSet3,cacheTagArray,cacheDataArray);
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	//cache miss tester
	printf("we wait for cache miss \n:");
	waitForCacheMissDelay();
	printf("waiting ended\n");
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	freeBothArrays(cacheTagArray,cacheDataArray);
	printf("******************MAIN ENDED SUCCESSFULLY
!**********************\n");
}

void testerFor_readByteAtAddress_writeByteAtAddress(){
	char ** cacheTagArray;
	cacheTagArray = loadCacheTagFromFile();
	char ** cacheDataArray=loadCacheDataArrayFromFile();
	//readByteAtAddress tester :
	//define global memory array :

	//THIS  WORKs :
	//1. readByteAtAddress is reading correctly the whole sbox and updating
the data array
	//2 .tag is showing - but for some reason the addresses that are
generated for the start of globalMem are always at set 3.
	//Creating the global memory (DRAM):
	int offsetInsideGlobalMemForSbox =0;
	unsigned char * globalMem =(unsigned char *)malloc(65536*(sizeof(unsigned
char))); //2^16 bytes .
	printf("the starting address for the globalMem: %u \n",&globalMem[0] );
	int offsetInGlobalMem =((unsigned int) globalMem )%16 ; //we want to be
aligned to a start of a block.
	putSboxOntoTheGlobalMem(offsetInGlobalMem,&globalMem[0]);
	//we read the whole sbox which is 16 bytes long - We should get 1 miss
and then 15 hits:
	int i=0;
	while (i<16){
	//printf("the offsetToStartFrom in globalMem is :
%d\n",offsetInGlobalMem);
	char byteFromMemory
=readByteAtAddress((int)&globalMem[0+i],globalMem,cacheTagArray,cacheDataArray)
;
	printf("Byte we read : %d \n",byteFromMemory);
	i++;
	}

	//Test for writeByteAtAddress: changing the first byte of the sbox to
0x99:
	//writing

writeByteAtAddress(0x99,(int)&globalMem[0],globalMem,cacheTagArray,cacheDataArr
ay);
	//reding back to see if the write was successful :
	unsigned char byteReadAfterChange
=readByteAtAddress((int)&globalMem[0],globalMem,cacheTagArray,cacheDataArray);
	printf("Byte we byteReadAfterChanging from cache(sould be 0x99)  : 0x%02X
\n",byteReadAfterChange);
	printf("Byte we byteReadAfterChanging globalMem(sould be 0x99)  : 0x%02X
\n",globalMem[0]);

	//preforming a read from the line after the sbox: (should give us a cache
miss:)

readByteAtAddress((int)&globalMem[0+i],globalMem,cacheTagArray,cacheDataArray);

	//freeing arrays and finishing Test
	UpdateCacheTextFiles(cacheTagArray,cacheDataArray);
	free(globalMem);
	freeBothArrays(cacheTagArray,cacheDataArray);
```

```
      printf("******************Test ENDED SUCCESSFULLY
!**********************\n");
}
```
