FINAL PROJECT:

Hardware Accelerator For Cache Timing Attacks Mitigations

Lidor Azani
Noam Hever

Supervision: Dr. Itamar Levi, Mr. Oren Ganon

18 minutes

What will we show?

1

Short introduction – overview of previous presentation.
The solution we chose.

2

Project's plan (waterfall) .
its development and analysis stages.
tools used.

3

The built solution:
Our Memory access flow – in software and hardware accelerated.

4

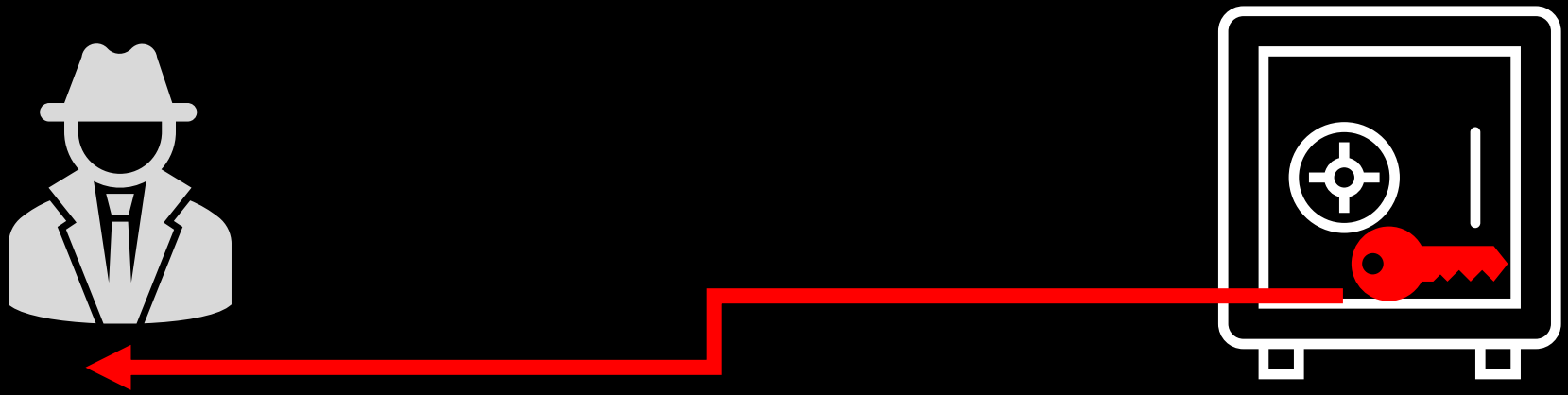
<u>Analysis:</u>

a. Cryptographic – statistical analysis.
b. Performance and acceleration analysis.

Introduction

- During previous presentation we talked about:

1. Side channels attacks .
2. Cache timing attacks.
3. Scramble cache – a defense mechanism.

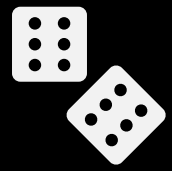


Information about the secret is leaking through a nonconventional information channel/signal.

Introduction

- Attacker can learn information about cryptographic secrets by observing access time patterns to the cache/memory on specific sets.

- Different Mitigations are focusing on preventing that by adding randomness to the memory/cache accesses .

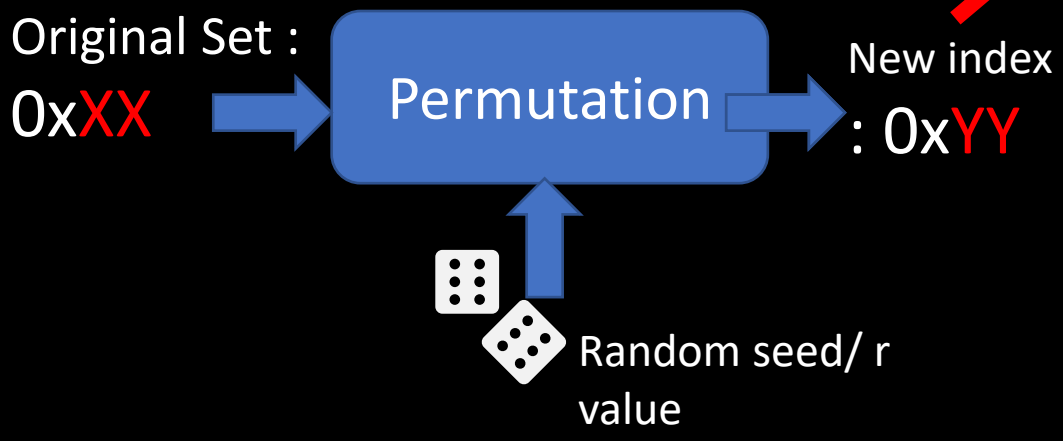


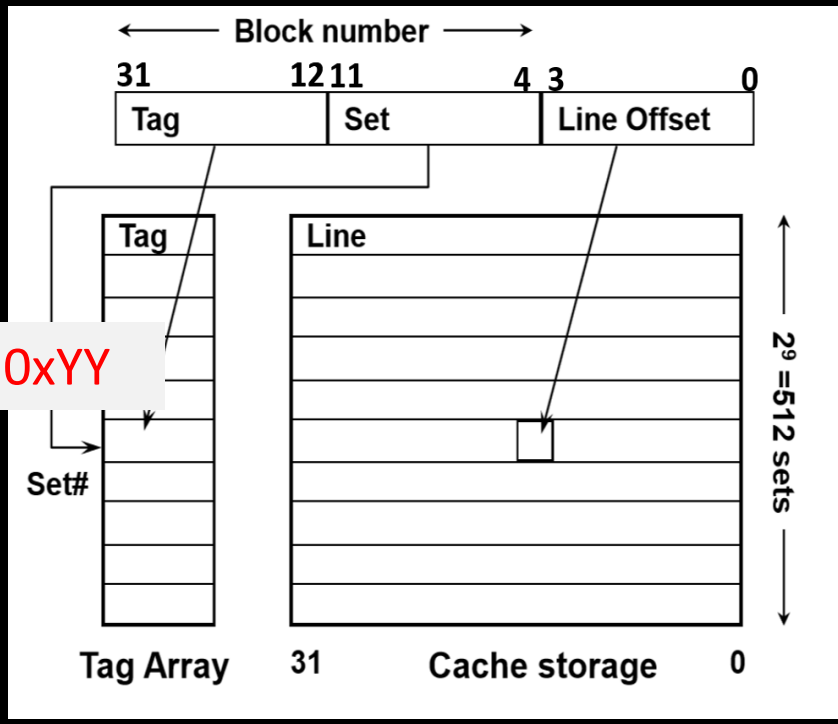
Introduction – A Reminder:

Random permutation cache – third solution

What can we do?
1. give each process a permutation table
Instead of using the "set" to decide on where we should bring the new data – we will decide using a permutation function on the value of the "set" in the address.

How do we calculate the mapping:

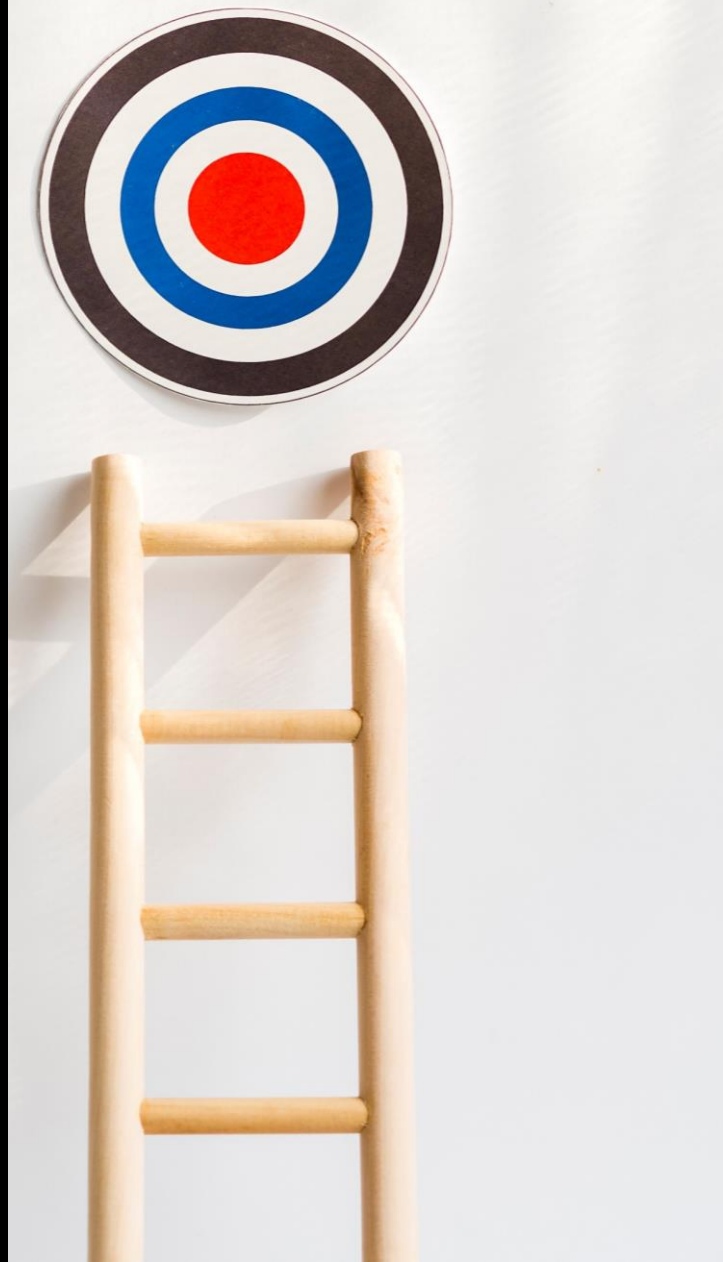




Introduction
Announced Goals:

1. Reduce the attacker's ability to learn anything about the memory accesses done by another process.

2. Randomize memory accesses in a cryptographically secure and efficient way .



2. Implementation

Inspired by the “Scrambled cache ” paper – with real implementation and no abstraction

2. Implementation

The "Scrambled cache " paper :

1. - only abstract emulation - no implementation
2. - no "full picture" of cost
3. - no effectiveness demonstration – with simulating attacks

Our "Randomized Memory area solution"

1. - Software and hardware implementation –usable by anyone!
2. - Cost analysis with a synthesizable processor
3. - Effective demonstration – ability to any C / C++ program – either attacker/Victim simulation or benchmarks

Amine Jaamoun, Thomas Hiscock, Giorgio Di Natale. Scramble Cache: An Efficient Cache Architecture for Randomized Set Permutation. Design, Automation & Test in Europe Conference & Exhibition (DATE 2021), Feb 2021, Grenoble, France. ff10.23919/DATE51398.2021.9473919ff. ffhal-03351626

2. Implementation: Development Stages:

Research papers

Planning software and hardware accelerations

C prototype

Validating: correctness, injectivity of permutation

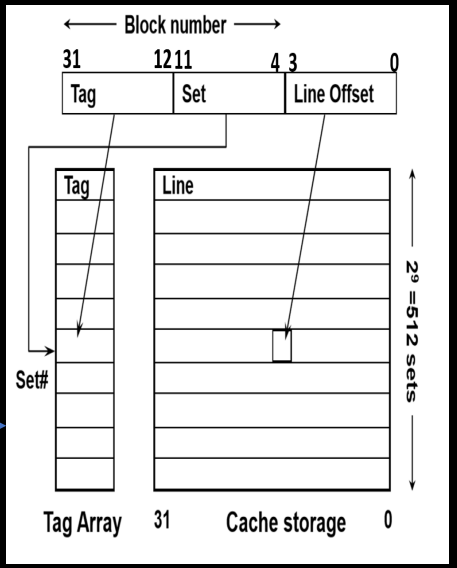
In TIE HDL – Accelerate

Measure, test and benchmark

Prove functionality, safety and efficiency

Solution schematics

Regular L1 Cache



Access memory with New address

History table – compare sets

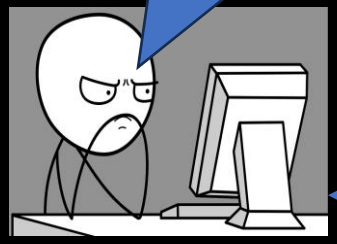
index	History table - original indexes
0	original set for memory line 0
1	original set for memory line 1
2	original set for memory line 2
3	original set for memory line 3
4	original set for memory line 4
5	original set for memory line 5

Permutation

New address

Original address

Read memory[10]



If sets are not matching: search history table for a match.
Then fix mapping by swapping lines in memory and update history table

On miss – access RAM

Memory area:

```
Memory arr:

line 000: 0x474F3000 : 0x00 0x00
line 001: 0x474F3010 : 0x00 0x00
line 002: 0x474F3020 : 0x00 0x00
line 003: 0x474F3030 : 0x00 0x00
line 004: 0x474F3040 : 0x00 0x00
line 005: 0x474F3050 : 0x00 0x00
line 006: 0x474F3060 : 0x00 0x00
line 007: 0x474F3070 : 0x00 0x00
line 008: 0x474F3080 : 0x00 0x00
line 009: 0x474F3090 : 0x00 0x00
line 010: 0x474F30A0 : 0x00 0x00
line 011: 0x474F30B0 : 0x00 0x00
line 012: 0x474F30C0 : 0x00 0x00
line 013: 0x474F30D0 : 0x00 0x00
line 014: 0x474F30E0 : 0x00 0x00
```

Finally ,Give the data to the program

2. Implementation

Software Model: Helping the programmer

Step 1: initialization:

The programmer defines a memory area as “randomized access” to protect sensitive data.

2. Implementation

Software Model: Helping the programmer

Step 1: initialization:

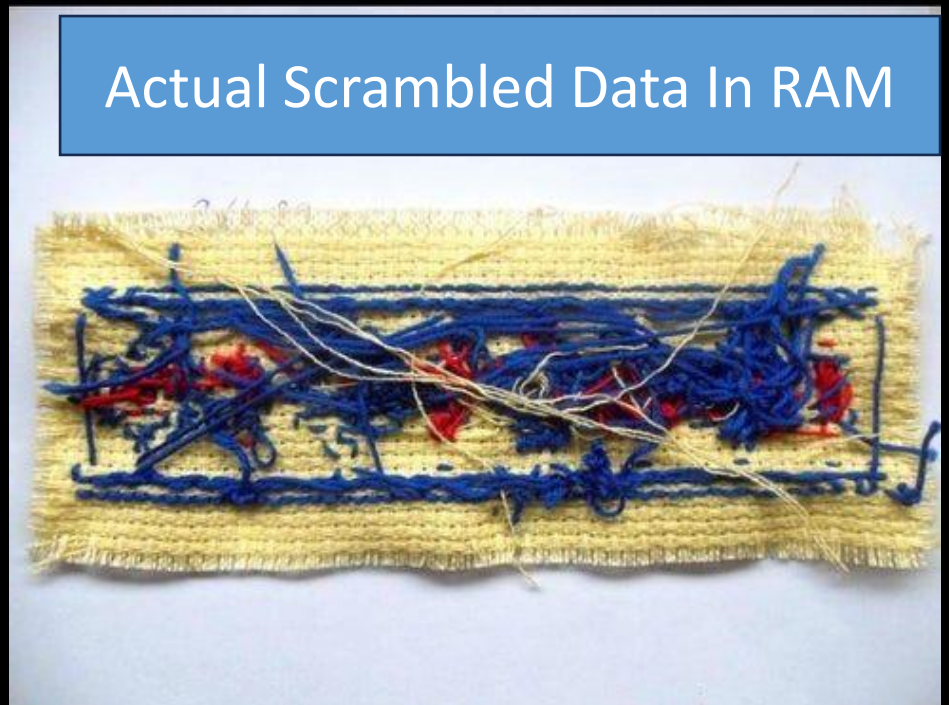
Note :
After the initialization- the programmer will access the memory as usual and is oblivious to the underlying permutation and actual “real” addresses.

2. Implementation

Behind the scenes:

A spin on a well-known meme

If the programmer has data in address 0xXXXXXX he can expect to read back the same data from 0xXXXXXX



2. Implementation

Memory access flow:

Current R –mapping rule

History table

Line index	Original set
0	0x6E
1	0x12
2	0x19
3	0x28
4	0xFD
5	0x41
6	0x56

Memory area:

```
Memory arr:

line 000: 0x474F3000 : 0x00 0x00 0x00 0x00
line 001: 0x474F3010 : 0x00 0x00 0x00 0x00
line 002: 0x474F3020 : 0x00 0x00 0x00 0x00
line 003: 0x474F3030 : 0x00 0x00 0x00 0x00
line 004: 0x474F3040 : 0x00 0x00 0x00 0x00
line 005: 0x474F3050 : 0x00 0x00 0x00 0x00
line 006: 0x474F3060 : 0x00 0x00 0x00 0x00
line 007: 0x474F3070 : 0x00 0x00 0x00 0x00
line 008: 0x474F3080 : 0x00 0x00 0x00 0x00
line 009: 0x474F3090 : 0x00 0x00 0x00 0x00
line 010: 0x474F30A0 : 0x00 0x00 0x00 0x00
line 011: 0x474F30B0 : 0x00 0x00 0x00 0x00
line 012: 0x474F30C0 : 0x00 0x00 0x00 0x00
line 013: 0x474F30D0 : 0x00 0x00 0x00 0x00
line 014: 0x474F30E0 : 0x00 0x00 0x00 0x00
```

2. Implementation

Memory access flow:

Programmer access an address within the array : 0xTTTTTSSO

2. Permutation on set bits – to randomize the cache line access

2. New address is : 0xTTTTTZZO

2. Implementation

Memory access flow:

2. New address is : 0xTTTTTZZO

3. Calculate line index in memory area $line\ index = \frac{New\ Address - base\ address}{2^{|size\ of\ offset\ field\ in\ bits|}}$

3.b. Check history table : Is the data we are looking for stored at the "New Address" line?

Is History table[line index] == original set ?

No!

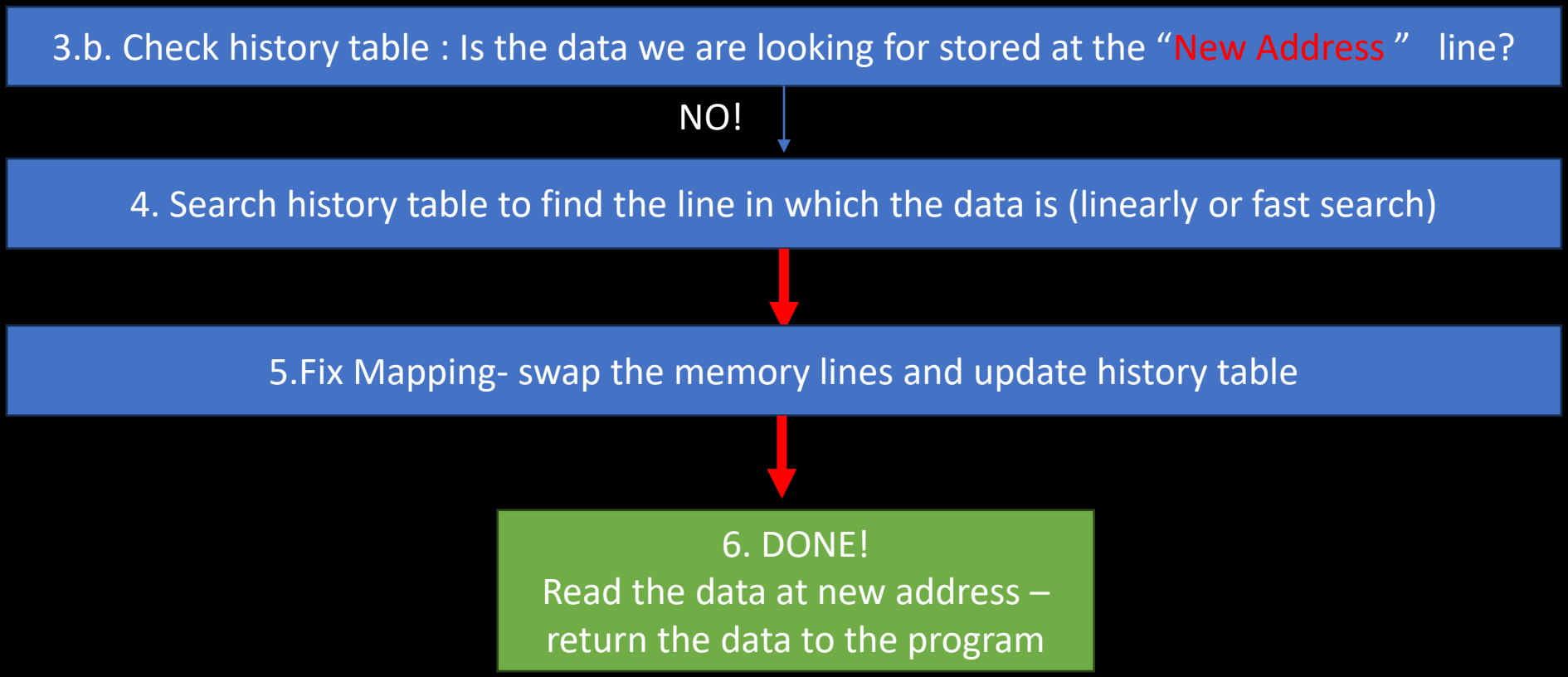
Next Slide

Yes!

6. DONE!
Read the data at new address – return the data to the program

2. Implementation

Memory access flow: Fixing the mapping



2. Implementation

Testing correctness and injectivity in C

Test clearBits ,Cswap2bit, Cswap8bit,

Test permute address for injectivity : 2^8 * 2^20

Test reading back memory from array : 2^8*2^20*2^2^12

Without permutation (R=0)

With permutation (20 bit R)

2. Implementation

Tie HDL acceleration

3 Tie operation

---

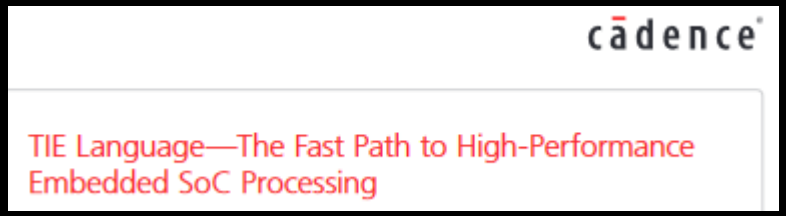
8 Tie functions

C_swap(conditional swap)

Line calculation

Address calculation

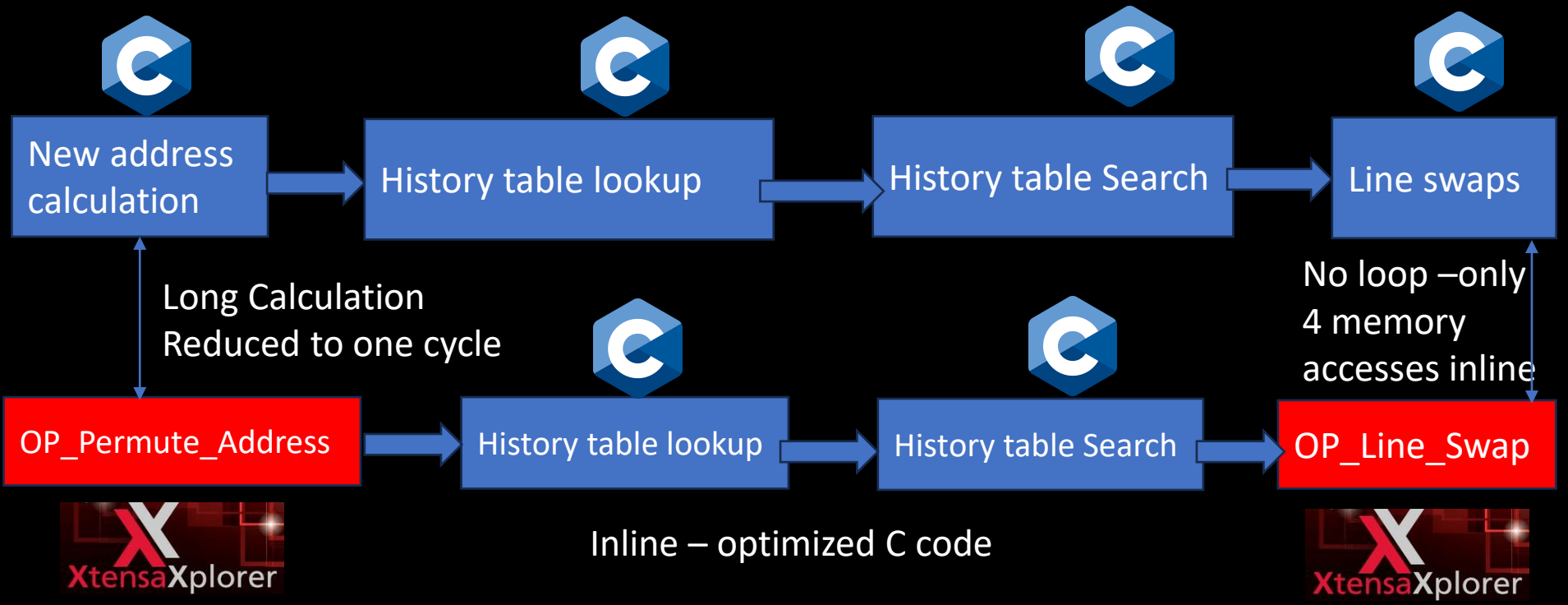
Vectorized 128 bit swap



Brute force testing :Comparison with C functions to verify correctness

2. Implementation

Tie HDL acceleration
Memory access flow speedup



Note: line index calculation, history swap and more were optimized by analyzing disassembly with Oren

3. Statistical Analysis

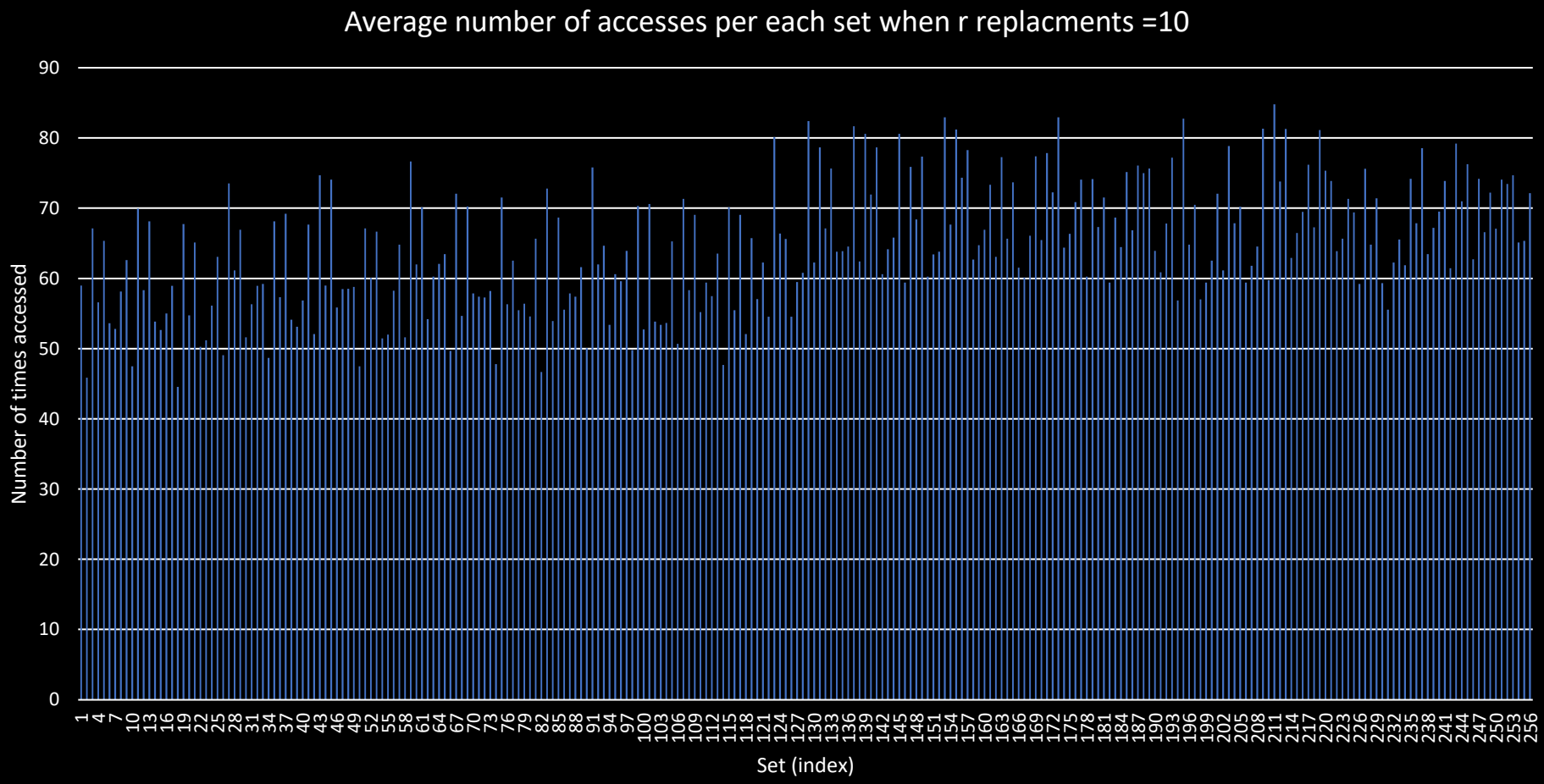
3.Stastical Analysis

We want to prove that :

1. Uniform distribution (cryptographic security) – attacker cannot learn from set accesses
2. Show set access pattern over time – see how it masks the “real ”set accesses

3.Stastical Analysis

- First test : Running a 15,000 sequential reads and counting how many times each set was accessed:



3. Statistical Analysis

- First test : Running a N=15,000 sequential reads and counting how many times each set was accessed:

- We used statistical tests to verify that the expected value for the average number of accesses per set in about n*1/256 .
- Coefficient of variation :of differences from mean
- As well as Pearson's **Chi-squared test** **to check the proportions are matching the expected 1/256 proportion.**

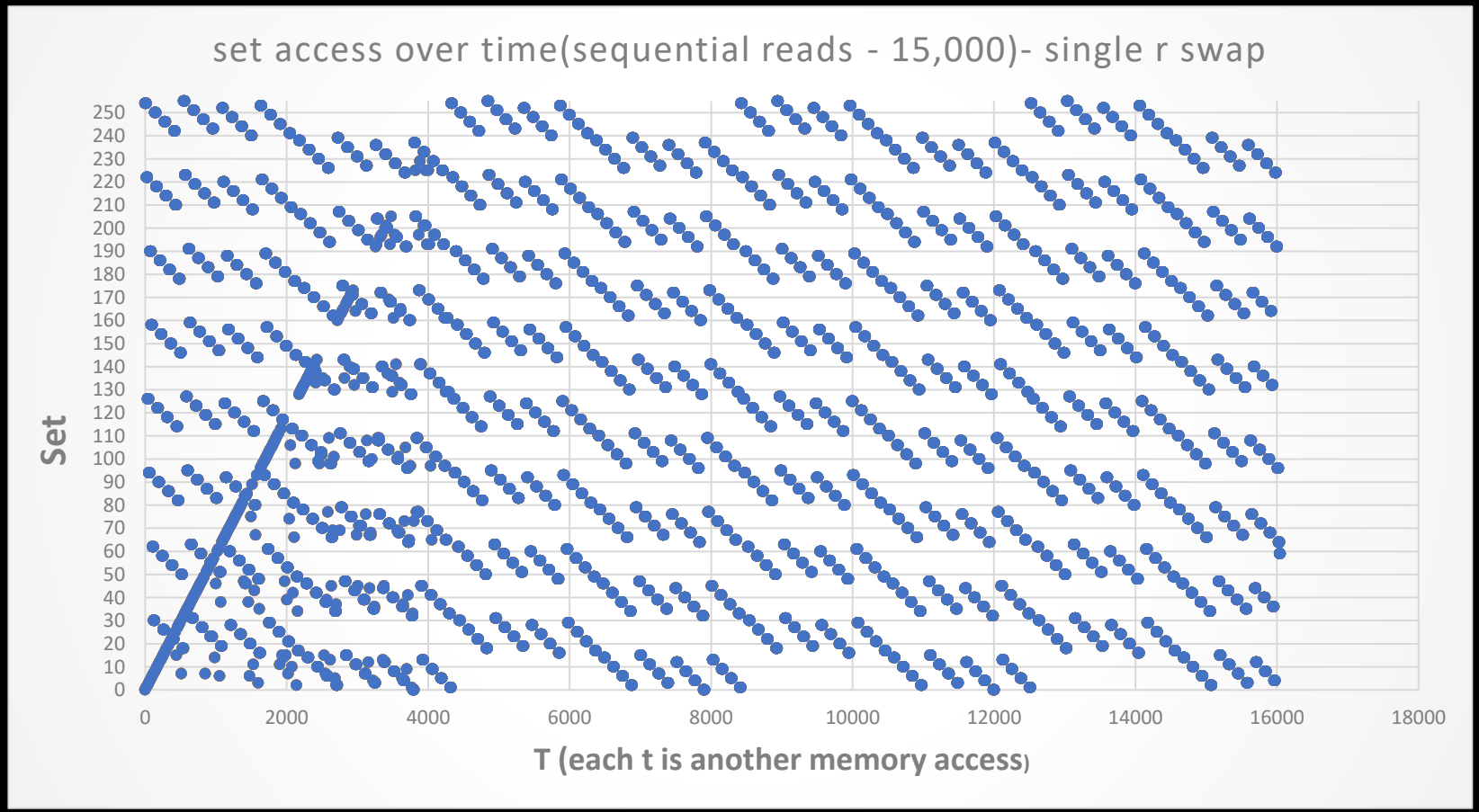
Set index

Experiment/Sample index out of N

sequent	s of r value					
Set index /number of accesses	*1*	*2*	*3*	*4*	*5*	*6*
0	81	86	38	40	52	67
1	51	69	24	23	38	38
2	81	69	38	68	52	66
3	67	68	22	67	38	68
4	81	69	38	38	52	69
5	52	57	24	23	42	52
6	65	70	12	52	54	40

3. Statistical Analysis

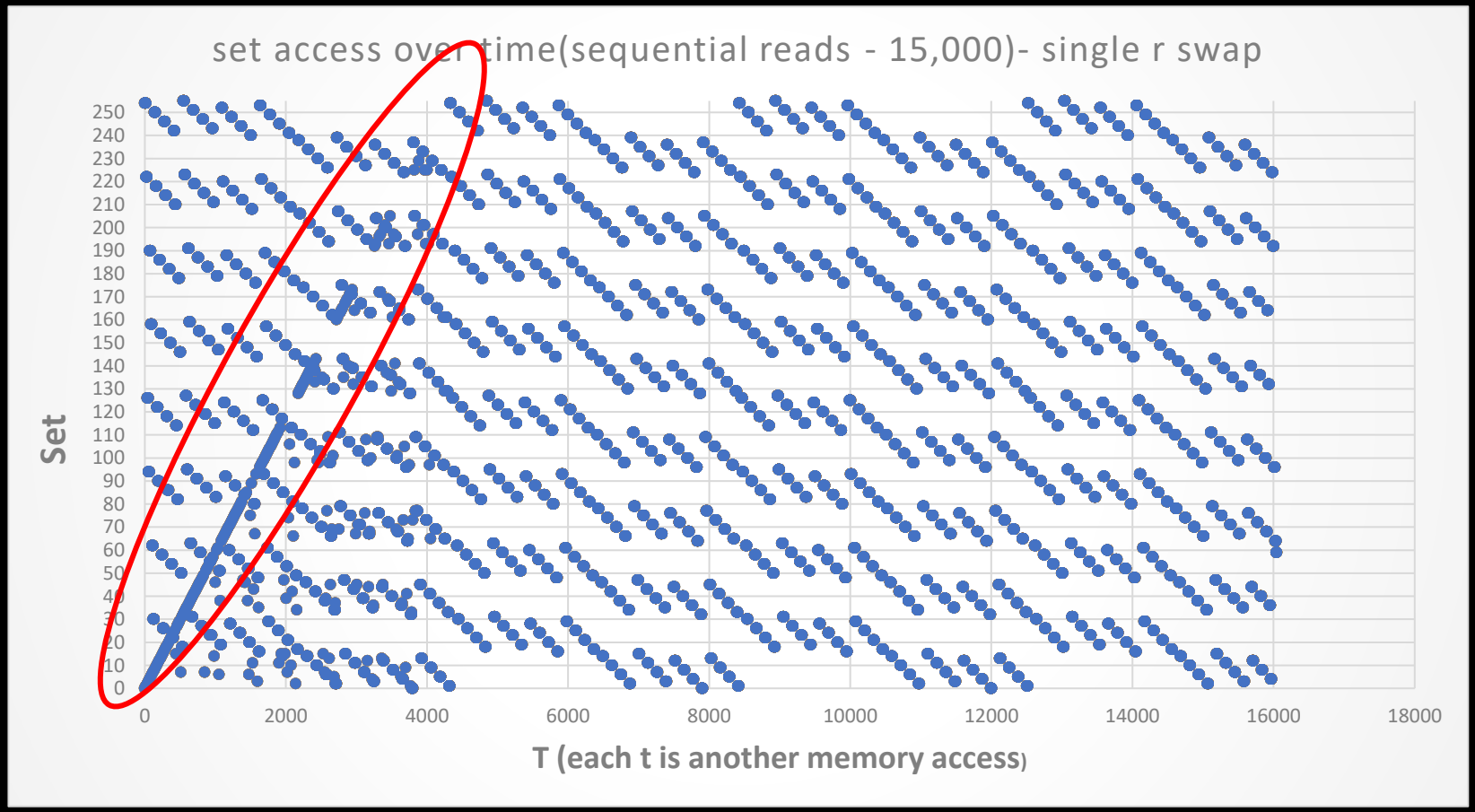
- Second test – How frequently should we swap the permutation?



3. Statistical Analysis

- Second test – How frequently should we swap the permutation?

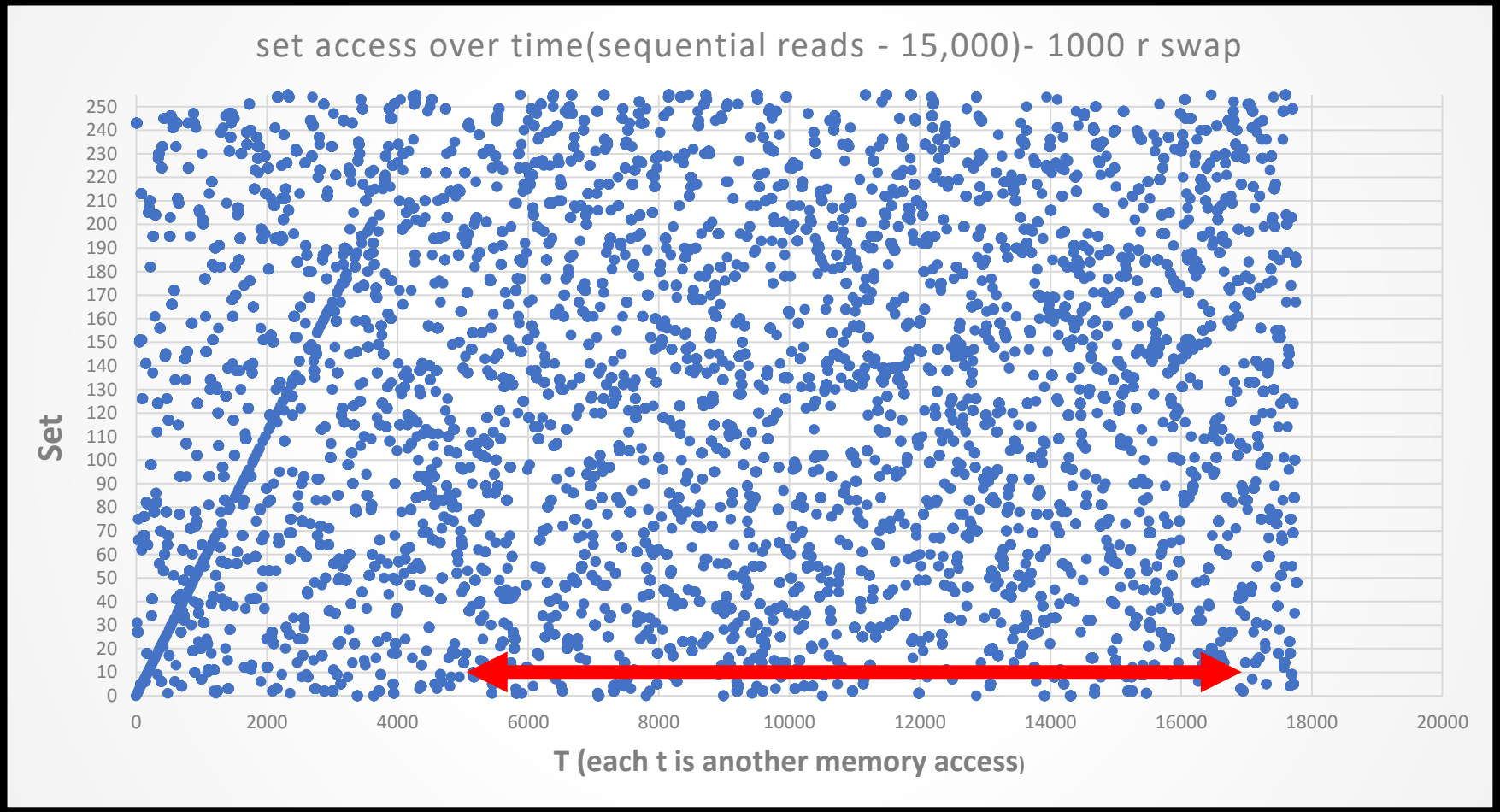
Pay attention to the linear line in the beginning:
Marks the initial shuffling of original memory mapping.



3. Statistical Analysis

- Second test – How frequently should we swap the permutation?

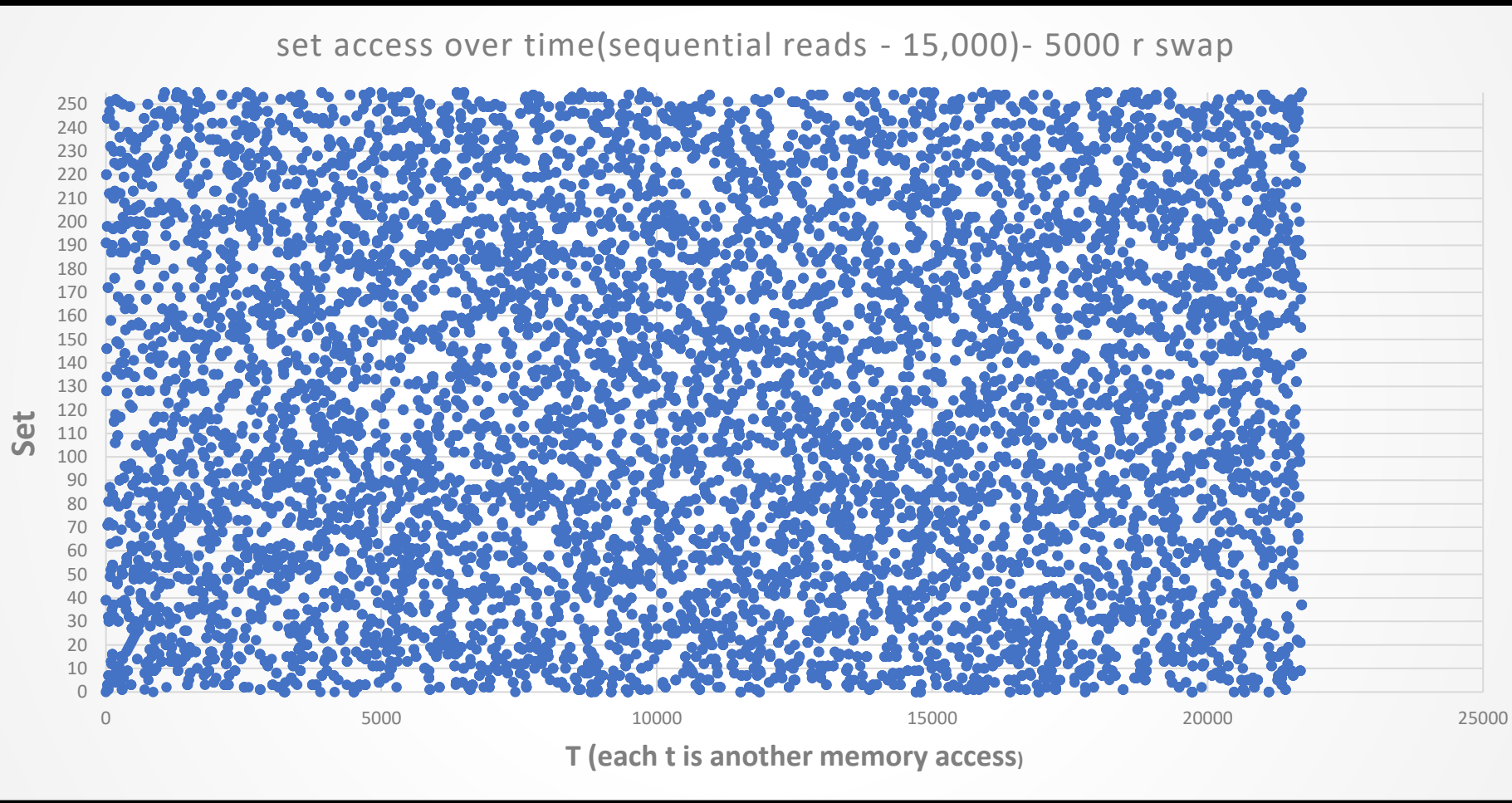
If we increase the frequency of changing R – the access pattern is not visible anymore



3.Stastical Analysis

- Second test – How frequently should we swap the permutation?

If we increase the frequency of changing R – the access pattern is not visible anymore

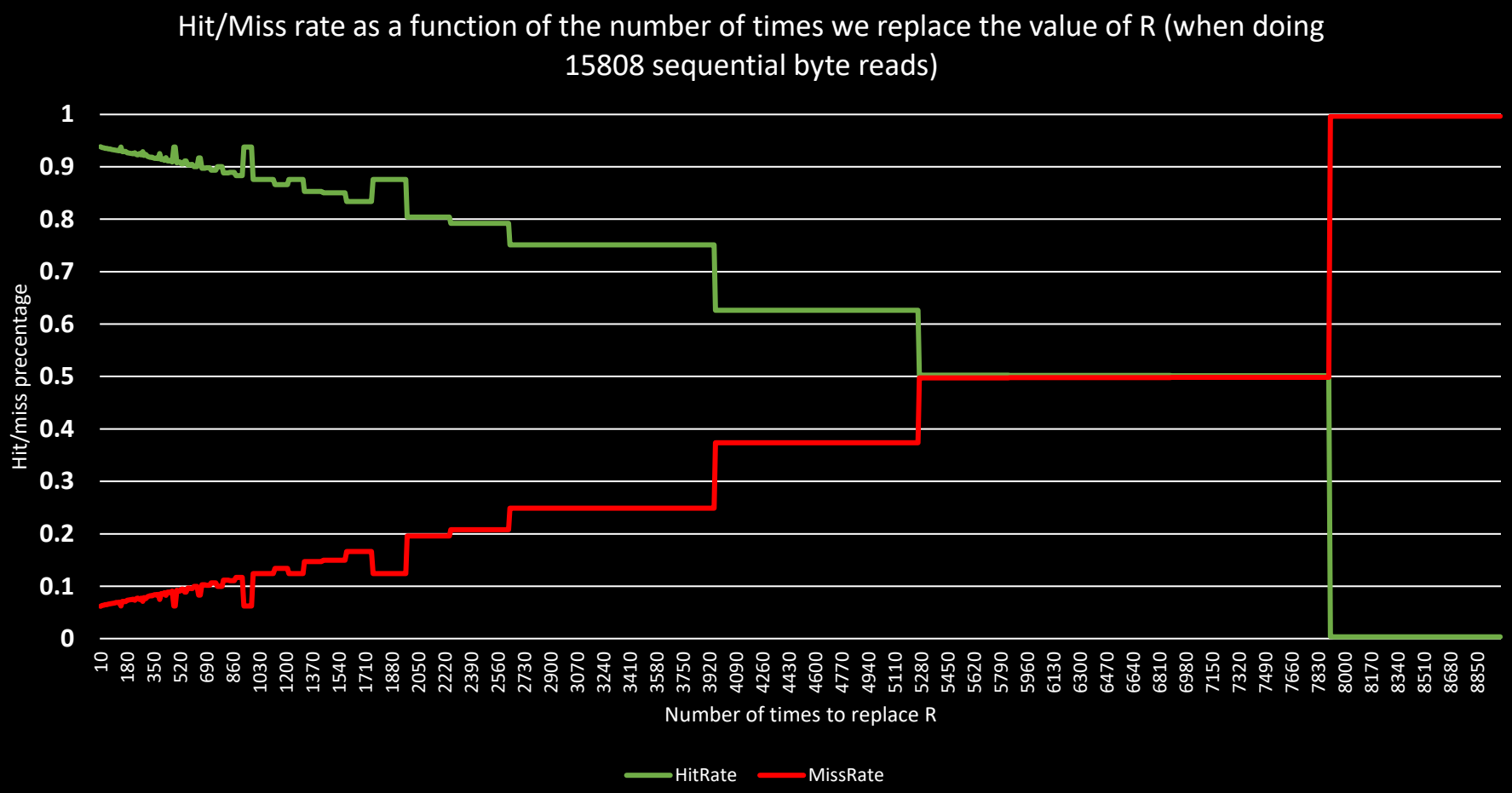


3. Statistical Analysis

- Third test –Protection performance penalty:

How many memory swaps will happen due to “misses” in history table-> when we increase the frequency of permutation swaps ?

(N=15,000 sequential reads)

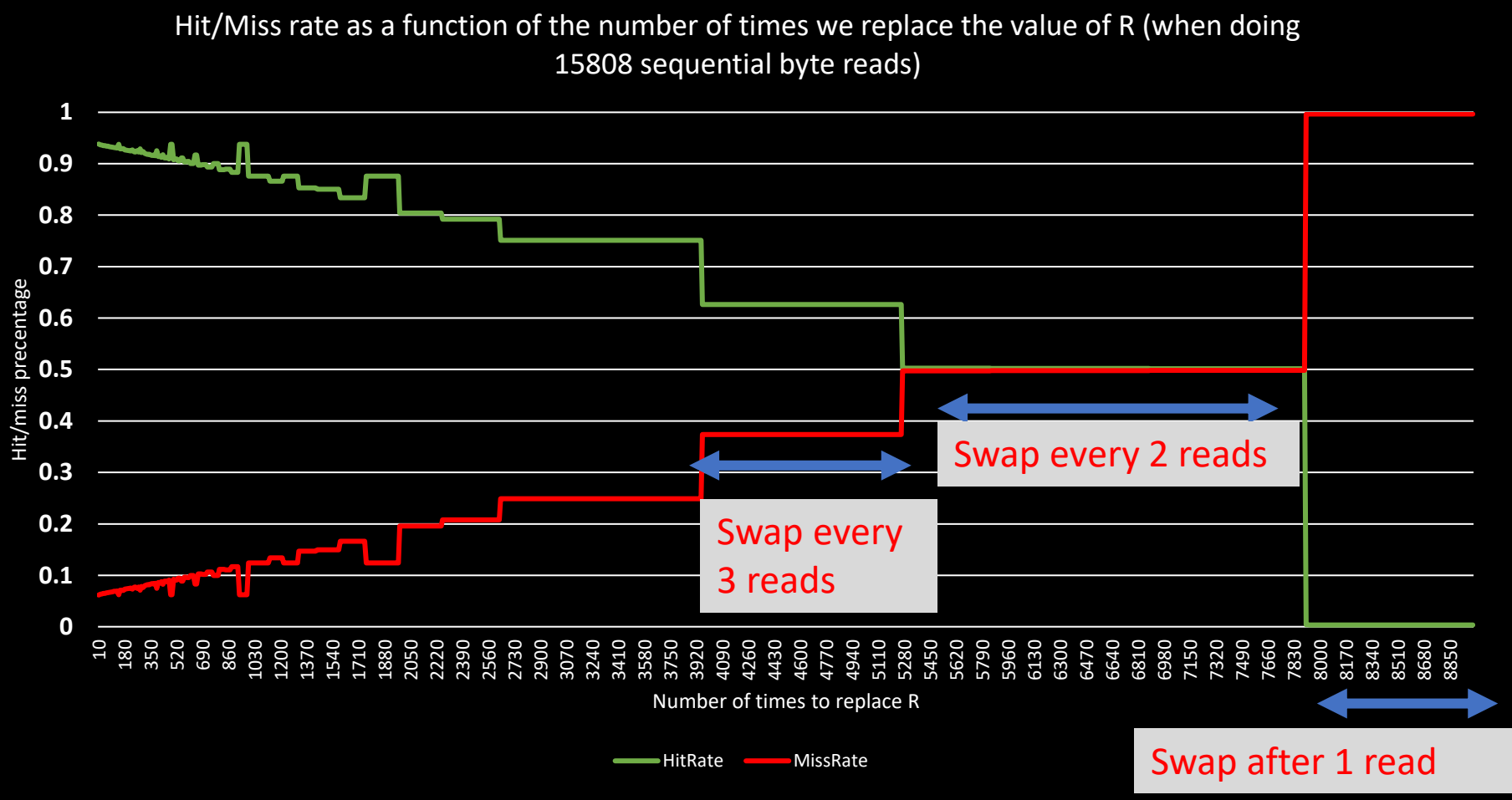


3. Statistical Analysis

- Third test –Protection performance penalty:

How many memory swaps will happen due to “misses” in history table-> when we increase the frequency of permutation swaps ?

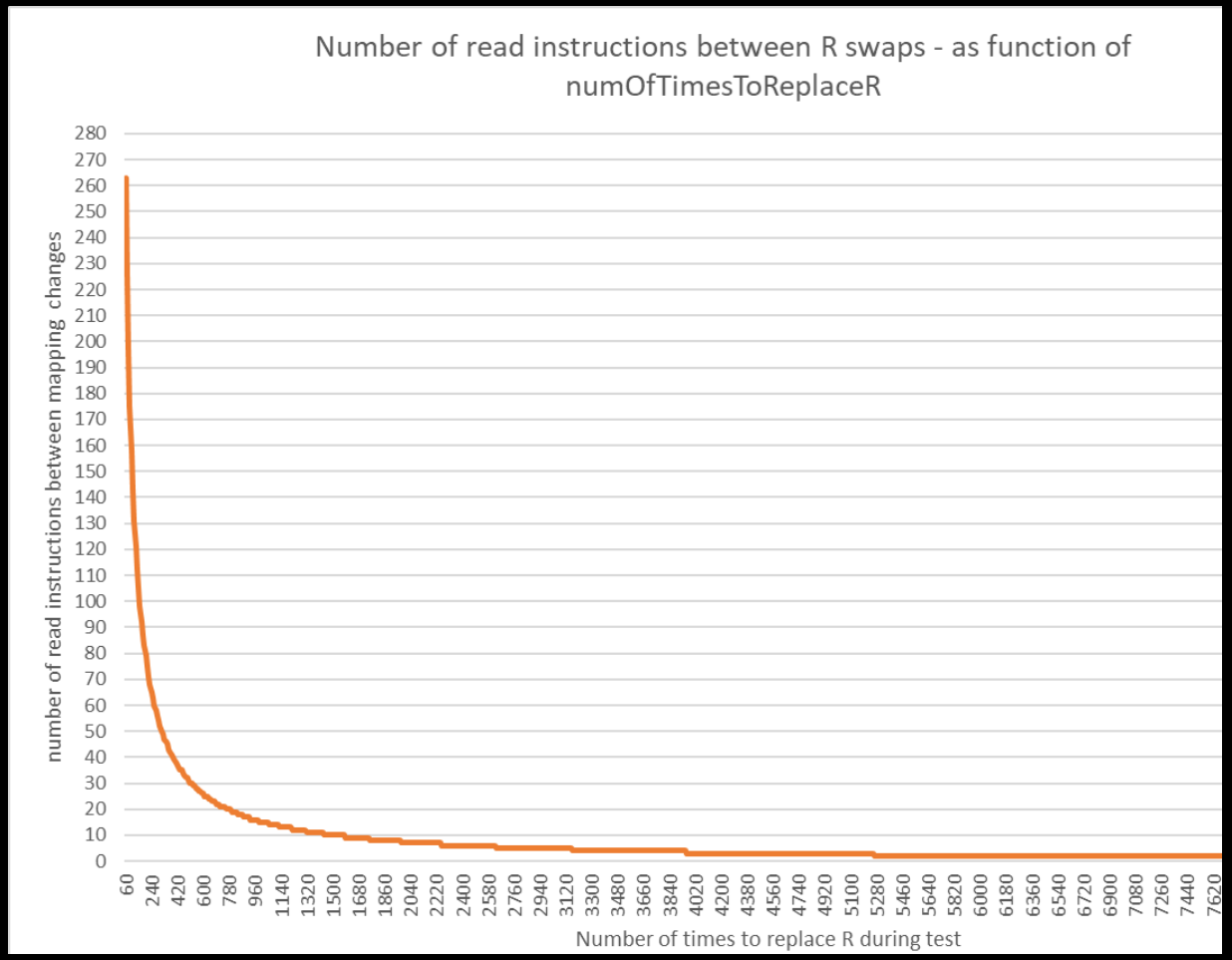
(N=15,000 sequential reads)



3. Statistical Analysis

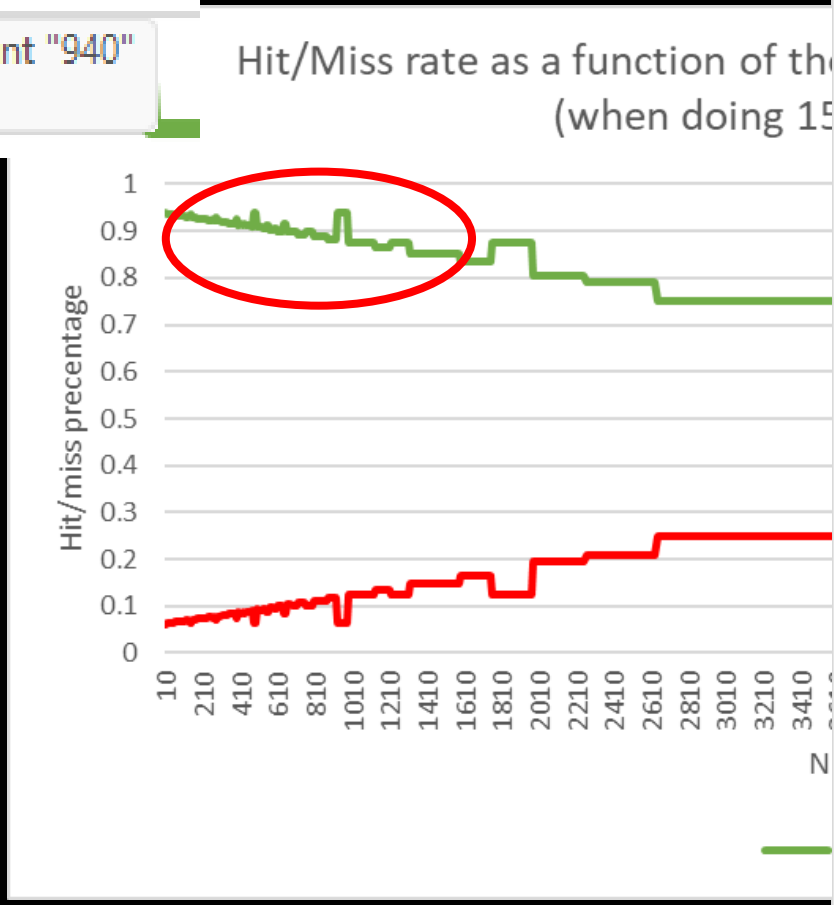
- Reaching a conclusion – how often should we swap the permutation for address mapping?

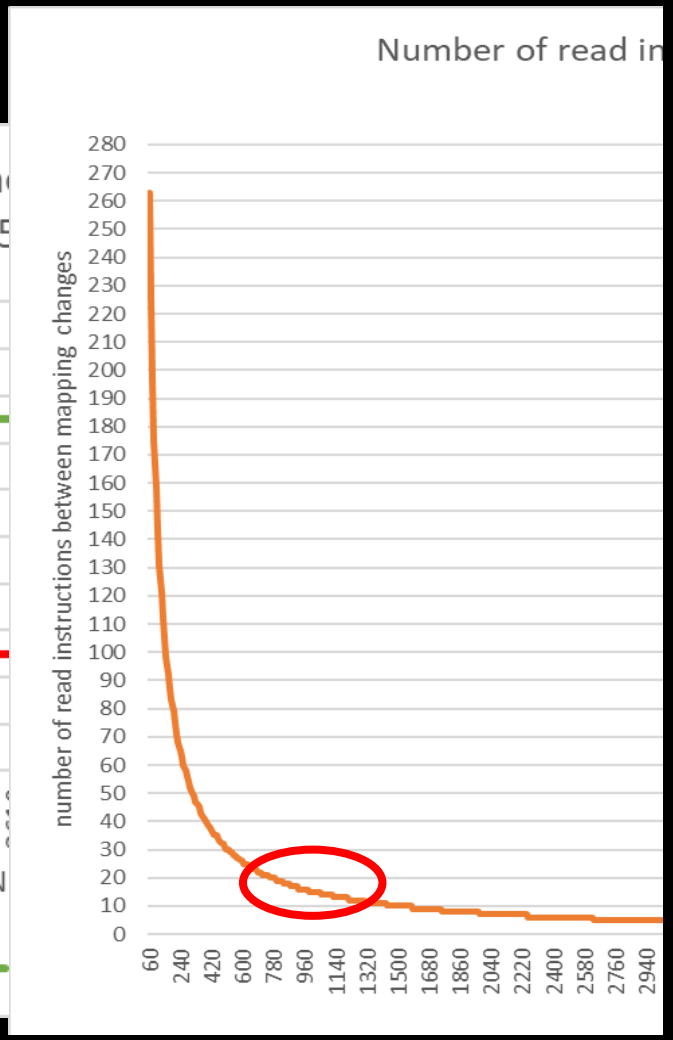
If We compare the Number of read instructions between R swaps - as function of numOfTimesToReplaceR .
We can answer the question : what is the optimal frequency range to swap the permutation (swapping R)?



3. Statistical Analysis

- Reaching a conclusion – how often should we swap the permutation for address mapping?





Conclusion :
In the range of 900-1000

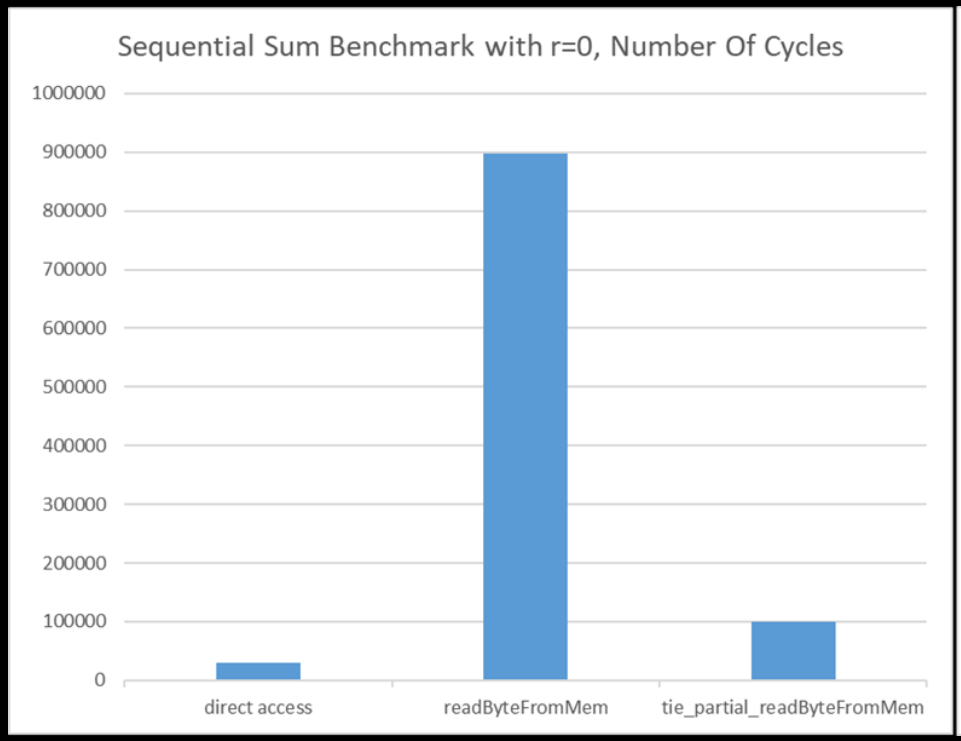
We get over 90% 'Hit'

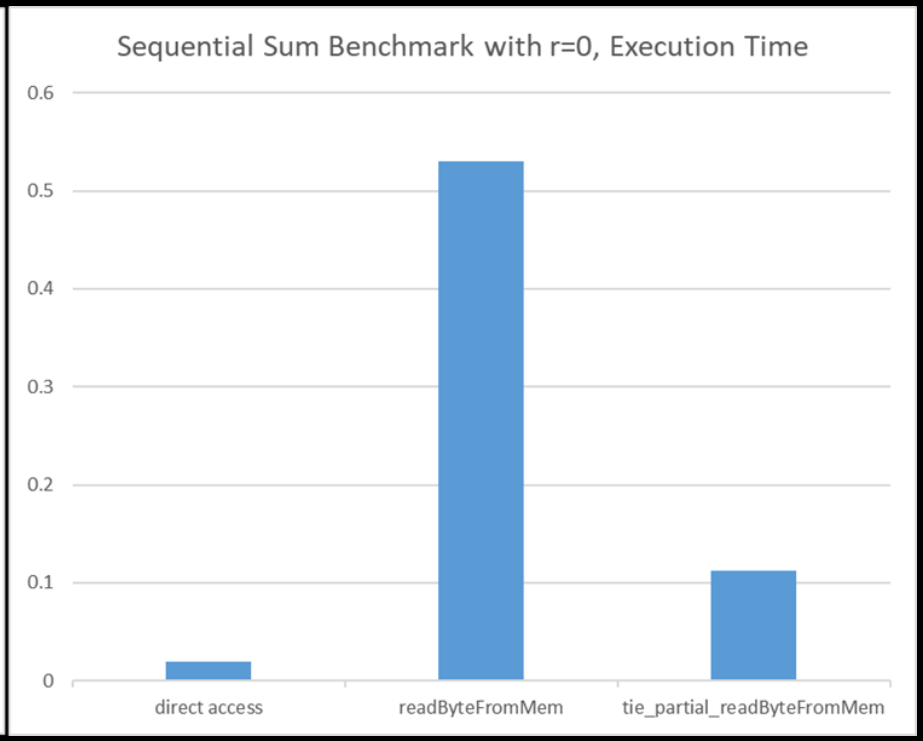
Which is an R-value swap every 15-17 memory accesses.

4. **Results and** Benchmarks

4. Benchmarks

1. Sequential Sum Benchmark: accessing whole memory
(4096 memory accesses)



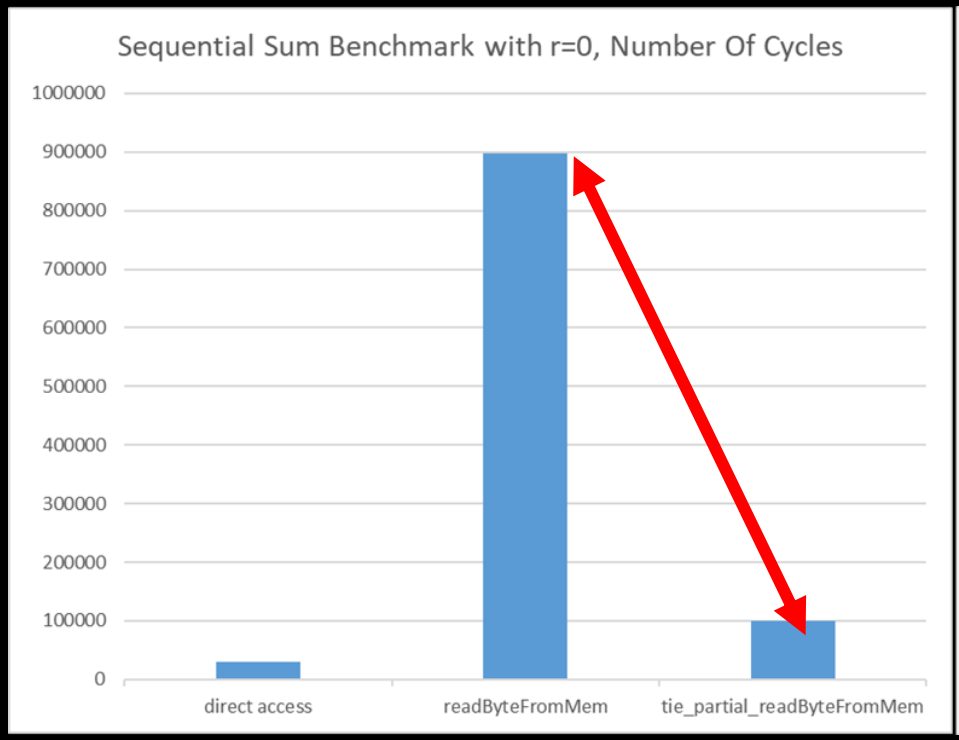


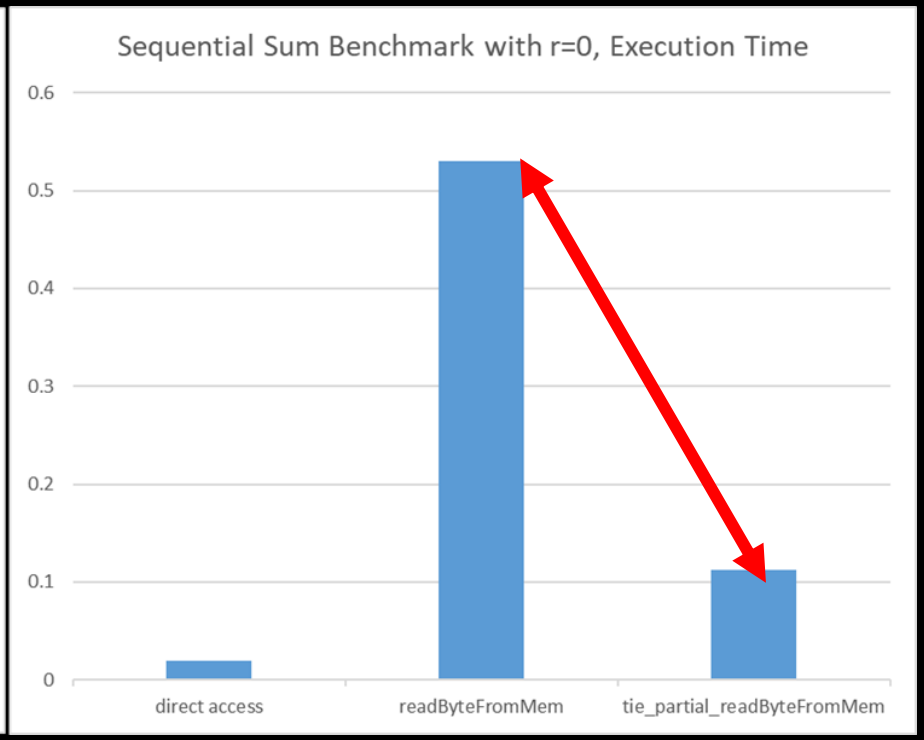
4. Benchmarks

1. Sequential Sum Benchmark: accessing whole memory

(4096 memory accesses)

Significant acceleration ! Tie operations saves us :





4. Benchmarks

1. Sequential Sum Benchmark: accessing whole memory
(4096 memory accesses)

Significant acceleration ! Tie operations saves us :

Cycles :

If R=0 -> Reduced by 89%

If R!=0 -> On average : Reduced by 60%

Due to history table searches and swaps

Time :

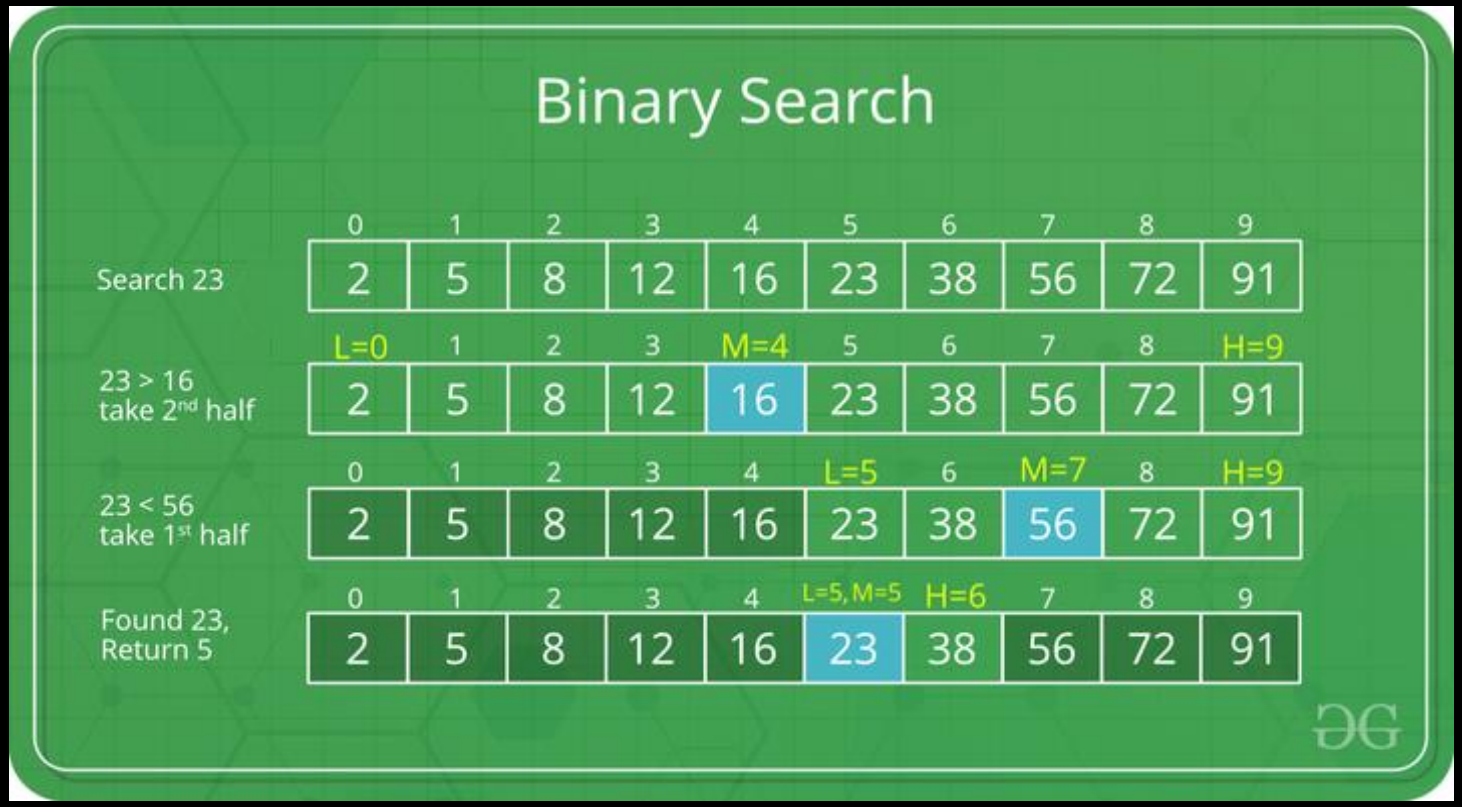
If R=0 -> speedup of 4.7 times faster!

If R!=0 -> average speedup of 2.3 !

4. Benchmarks

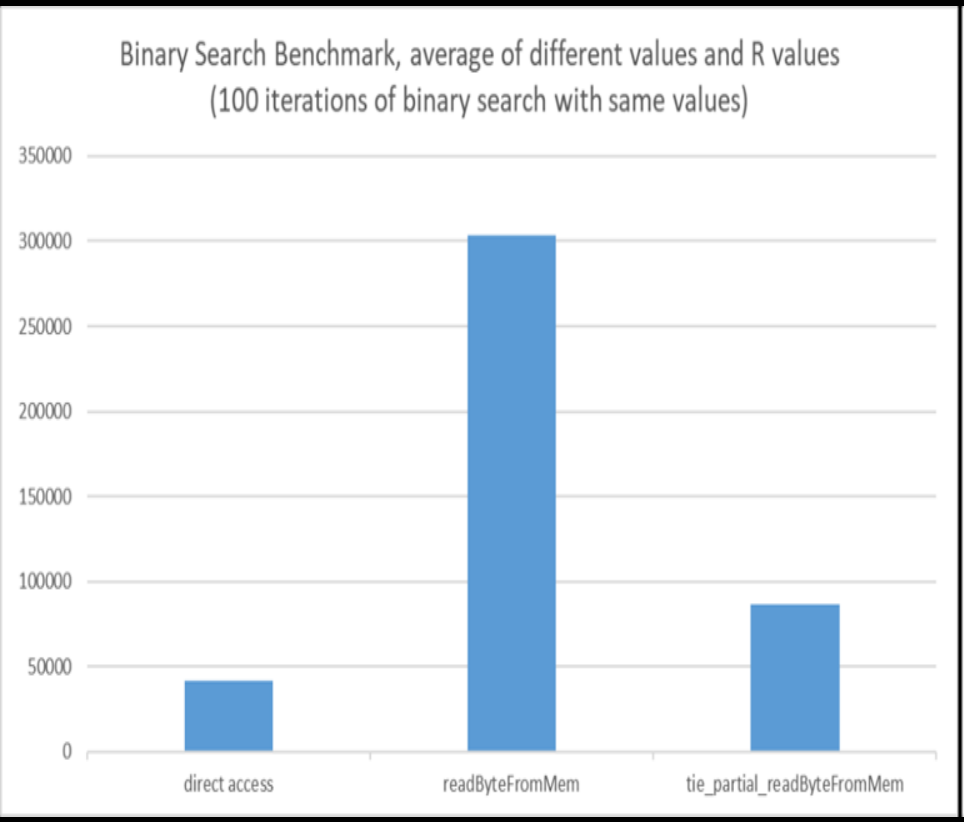
2.Binary Search Benchmark-Results

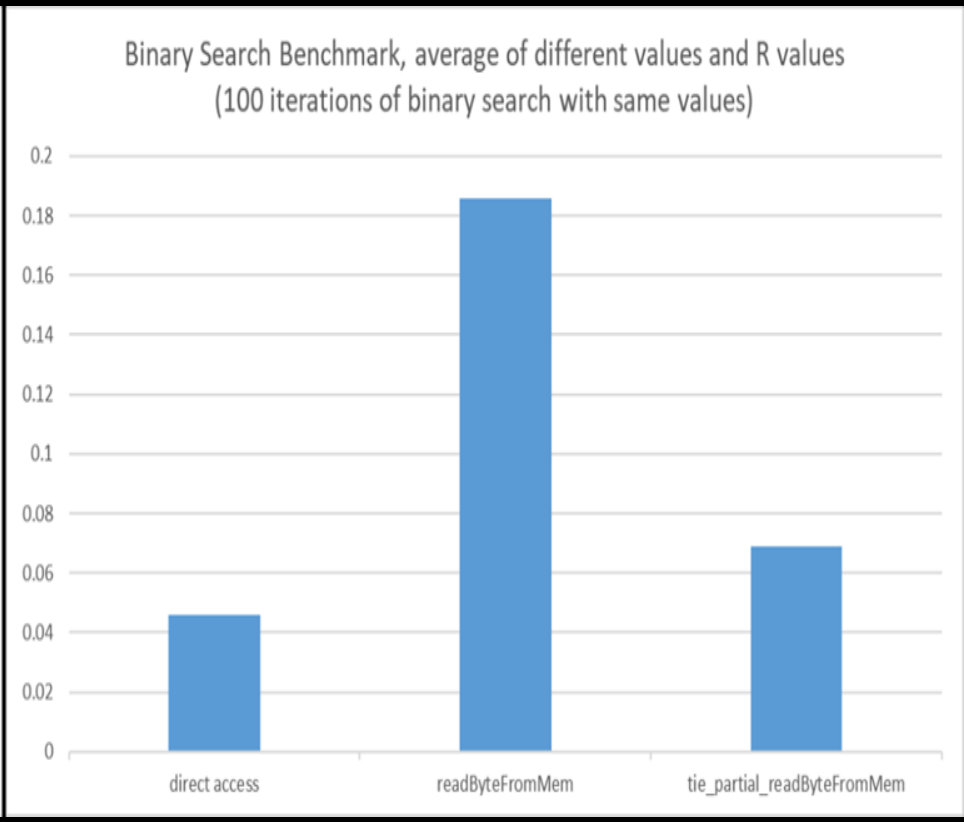
Non-sequential reads benchmark :
unknown access pattern



4. Benchmarks

2.Binary Search Benchmark-Results





4. Benchmarks

2.Binary Search Benchmark-Results

Significant acceleration ! Tie operations saves us :

Cycles :

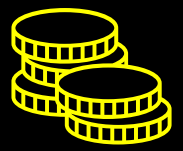
If R!=0 -> On average :
Reduced by 72%

Time :

If R!=0 -> average speedup of 2.7 !

Time Penalty

- Disadvantages, downsides and cost of using our information leakage protection:



- If a programmer deploys our solution :the average runtime for sequential 4096 reads will increase by up to 14 times (on average).

- If a programmer deploys our solution – the average runtime for Binary search will increase by 1.5 times. The cycle count will be 2.1 times higher

Conclusion

- The system works

- It protects from leakage by making “SET” cache line accesses randomized (0 = 1/256) .

- TIE HDL gives a significant acceleration to our algorithm

- Programmers should choose wisely when to implement this solution – only for critical safe/cryptographic data – to avoid the performance penalty.

שאלות?

תודה רבה