הפקולטה להנדסה
המעבדה לחומרה בטוחה

Hardware Accelerator for Cache Timing Attacks Mitigations

נעם חבר -
לידור עזאני -

פרויקט שנה ד' לקראת תואר ראשון בהנדסה

מנחה: אורן גנון
מנחה אקדמי: ד"ר איתמר לוי

אוקטובר 2023

תקציר:

אבטחת מידע הפכה להיות עניין חשוב בכל רכיב חשמלי בימינו, האקרים שמים להם למטרה כל מוצר שיש בו מעבד ומנסים לדלות ממנו מידע סודי על התוכנית שרצה.

מאז שנות ה-2000 עלה הנושא של מתקפות ישירות על החומרה עצמה ולא רק בצד התוכנה, אשר מטרתן לדלות מידע סודי על ריצת התוכנית ע"י בחינת החומרה וניצול ידע קיים על ארכיטקטורת המעבד. מתקפות כאלה נקראות "מתקפות ערוצי צד" והן מתמקדות בפרמטרים פיזיים, כמו למשל מדידת זמן ריצה, צריכת הספק, דליפה אלקטרומגנטית ועוד.

אמצעי הגנה שנועדו למנוע מתקפות ערוצי צד כרוכים בד"כ בתקורה משמעותית, בין היתר בהיבט של האטת ביצועים. מטרת המאמר היא להציע אמצעי הגנה נגד מתקפות מדידת זמן של המטמון (cache). במאמר זה אנו עושים שימוש במעבד של חברת Cadence בו סט ההוראות (ISA) ניתן להרחבה, וכך ע"י שימוש בשפת חומרה שנקראת TIE אנו יכולים לשלב פונקציות בשפת C עם פונקציות בשפת TIE במטרה להאיץ בצורה משמעותית את זמן הריצה ומחזורי השעון הנדרשים על מנת ליישם את אמצעי ההגנה שמימשנו.

אנו נתמקד בתרחיש בו יש מעבד פשוט יחסית עם זיכרון מטמון (מעבדי embedded) אשר מריץ מספר תהליכים במקביל או לסירוגין, למשל בעזרת RTOS או מערכות הפעלה low level אחרות. למעבד יש הגנה בסיסית/סטטית שמונעת גישה לא מורשית מחוץ לאזור הזיכרון של כל תהליך.

תהליך אחד הוא "תוקף" אשר מנסה לדלות מידע סודי על ריצת התהליך השני, ה"קורבן". הוא מנסה לעשות זאת על ידי בחינת שינויים בזמני ריצה / גישה לזיכרון הנגרמים בשל פינויי מידע מזיכרון המטמון, כך שהמעבד נדרש להביא את המידע למטמון מהזיכרון הראשי (RAM) – פעולה שלוקחת הרבה יותר זמן (בד"כ איטי בשני סדרי גודל, משתנה בין מעבד למעבד).

התוקף ניגש למידע ספציפי בזיכרון כדי להשפיע על המידע שנמצא בזיכרון המטמון, ואז מודד את זמן הריצה של הקורבן או את זמני הגישה לזיכרון שלו בעצמו, במטרה לבדוק האם המידע שהוא צריך כבר נמצא במטמון או שאולי המידע כבר פונה ממנו כיוון שהתוקף כבר ניגש אליו בעבר. באופן הזה הוא יכול לדלות מידע על פעילות התהליך ה"קורבן" - ולגלות סודות קריפטוגרפיים כמו מפתחות הצפנה ועוד.

ע"מ למנוע זליגת מידע דרך ערוץ הצד המדובר - הוצעו פתרונות רבים לאורך השנים. המשותף לכולם הוא "מיסוך" של המידע הזולג דרך מדידת שינויים בזמני ריצה.

הפתרון שבחרנו נקרא "Randomized Memory Area" או RMA - קביעת אזור זיכרון בו הגישה לזיכרון בפועל מרונדמת מאחורי הקלעים בעזרת פרמוטציה עם מפתח סודי - כאשר המיפוי משתנה באינטרוולים קבועים. מה שגורם לכך שאנו "מערבבים" את הכתובות בהן המידע שמור בזיכרון בפועל - כך שהגישות לזיכרון בפועל-אלו אשר גורמות לשינויים במצב זיכרון המטמון - מרונדמות . לכן, לא ניתן יותר ללמוד מידע מערוץ הצד המדובר.

יצרנו את הפתרון גם כספריית קוד בשפת C הניתנת לשימוש על כל מעבד. לאחר מכן יצרנו האצת חומרה המחליפה קוד ארוך בשפת C לפקודות אסמבלי המתבצעות במחזורי שעון בודדים. ניתוח על הביצועים ויתרונות ההאצה יובאו בהמשך

Hardware Accelerator For Cache Timing Attacks Mitigations

Lidor Azani and Noam Hever- Bar Ilan University, Engineering faculty, Israel.

Abstract: **In literature there are many studies regarding side channel attacks, specifically cache timing attacks that use the fact that the cache is a shared resource, to learn secret information that belongs to the 'Victim' process. Some mitigations were suggested throughout the years - but some of them were proven to not provide sufficient security or they were not implemented on real hardware and only shown as an abstract scheme. In this paper we introduce our "Randomized Memory Area" mitigation for side channel attacks scheme.**
The mechanism is targeted at embedded systems with cache memory. It is based on memory shuffling and permutation and does not require changes to cache IP HDL blocks.
It can be used in a slower software-only mode (C library). Alternatively, it can be used with hardware acceleration and specialized instructions on a processor that supports it . We have created the software and a low-cost and efficient hardware acceleration on a synthesizable Cadence Tensilica Xtensa processor. Both are presented and analyzed in this paper.

I. Introduction

Today almost all CPUs have on-chip cache memory. Cache memory is a smaller memory on the chip that can have much shorter access times compared to accessing the RAM. The data is stored in cache using the time and space locality principals - assuming that data that was recently accessed will be accessed again in the near future (time locality) and that the data saved in the same "block" around the data (arrays, structs, objects etc..) will be accessed in the near future.
Sometimes there are multiple levels of Cache (L1,L2,L3) and in multi core CPUs usually each core has its own L1 cache but in all cases there will be a level of cache that is shared amongst all cores threads and processes. Thus, the CPU cache is a "shared resource" for all of the system and programs and processes running on the CPU.

In the 90s , on chip CPU caches became mainstream and present on most computer systems from home PCs to Enterprise Servers systems.(although the concept and implementations existed since the 70s). Since then, researchers began to look into the security vulnerabilities [1][2][3] that arise from the existence of a shared cache resource in a computer system. The cache is effectively creating a covert /side channel in which information is passed about the actions of each process and its memory access patterns.
This type of side channel is called "Architectural side channel" [1] which means that the attacker is using his knowledge/public knowledge about the architecture of the CPU to find a side channel that leaks information about processes running currently or previously on the CPU.

There are multiple “Architectural side channels”. Some are using traces left behind after calculations in the FPU unit [2] or in the TLB [2] and more [1][2][3].

In our case we are focusing on “Cache timing attacks ” [1][2][3] many cache timing attacks exist across multiple scenarios of attack. In some scenarios the processes are running in parallel. [1][2] such attacks are “prime and probe” and “flush and reload” variants. Other attacks are using a scenario in which the attacker process is using/invoking/calling the “Victim” process. For example, calling a shell command/service for encryption. Some examples are “evict and time” and “cache collision attack” on AES encryption .

The information that the attacker can learn varies between different scenarios. In some cases(like “prime & probe “) the attacker can only learn information about the “Eviction Set” of the addresses used/accessed by the “Victim” process . (the “Tag” field) [1]
On its own, it is not a very powerful attack. But, when used as a part of sophisticated attacks - the information is used in some stage of the attack to take advantage of a vulnerability. [5]
In other cases, the attacker can learn information about the values of inner variables of the “Victim” process such as secret keys and program flow /decisions. For example, in the “flush & reload” scheme the attacker can learn the whole e-symmetric cryptographic key. And in the “cache collision” attack the attacker can learn some or all of the bits of the symmetric key.

Some mitigations for cache timing attacks were offered throughout the years. [1- page 18 -table 6][2][3] [link to table in overview paper].
A few notable mentions are:

1. **Hardware cache partition** - either splitting the cache sets between the different processes or creating a hardware mechanisms to assign locking attributes to every cache line, allowing sensitive data such as AES tables to be selectively and temporarily locked into the cache. [1- page 16]
2. **Cache flushing** - whenever there is a context switch (the OS switches between processes) flush the whole L1 cache (and any local state affected by the previous process) - is a very costly measure- but can be turned on selectively - meaning that a process can demand that whenever it is being switched and taken off the CPU - the system must “flush the cache” - in order to not leave any traces of its internal state behind . [1-page 21]
   (disadvantage- doesn’t affect attacks that focus on total runtime differences based on prior cache state- but can be solved with flushing before and after). [1]
3. **Injecting Noise** - through random cache fills by randomly adding memory accesses and reloads of memory. (Disadvantage - according to the paper - adds 100-120% runtime penalty). Adding random evictions/flushes or cache misses. (for example - we can decide that some of the memory accesses that will ignore the cache - or evict instead) [1- page 20 ‘fuzzy time’’]
4. **Random permutation cache**- (RPcache) - instead of processes evicting each other when they access memory addresses with the same set/index, each process will have a permutation table that will store the mappings between its own sets to cache lines.

Each line in the cache will store an ID value that is associated with the process that used it. When an eviction of cache contents from different processes is needed (if the cache is full) then the evicted line wouldn't be equal to the address' set but to a random selection of cache set to evict - and updating the permutation table with the result-
(disadvantage- requires a lot of memory - and slows down cache access because we need to read from the permutation table before we can look for the information in cache).[1][8-page 396]

5. **Cache coloring** - using the range of bits that overlaps between the range of bits used for the cache set and for the frame (virtual address decoding for page walks) -to establish "pools" of segregated memory regions that cannot overlap between processes. The aim is to provide stealth pages for storing security-sensitive data, such as the S-boxes of AES encryption. (Advantage- successful in experiments, but is difficult to implement due to differences between how different architectures are dividing the address, and also differences within architectures, such as x86, between processor generations that require an overhaul of the calculation of where these separate memory pools are). [1 -page 18]

6. **Holding sensitive data in registers** - either by using existing or dedicated register file/vector for storing the information. This will be effective in preventing cache collision attack on AES - as the access time to the register will be deterministic - and won't affect cache state. But this change would not help much against any other attack. (flush+reload, prime +probe etc.)

2.Background

One of the mitigations suggested in papers is the "Scramble cache" [4] which is an improvement upon previous "**Random permutation cache**" ideas.
In the paper, the authors created a simulation of a CPU cache, but changed its logic and the way the cache is accessed. Instead of the usual "N-way associative cache" which is based on direct mapped caches alongside one another. Usually we will access the line corresponding to the "set" of the address and compare the tag value stored in the corresponding line -to the tag value of the address.
If there is a match - then we have a cache hit - and we can read the data from the cache without costly RAM access. If it does not match - we need to evict the line from the cache and bring the appropriate line from RAM to the cache (locality principle).

In the "Scrambled Cache" we are permuting the set field of the address before we are checking if the tag value matches. Therefore, making cache line accesses different than the "Set" of the address. If we swap the permutation often - then we will be able to hide/mask the accesses to the set.
[4].
This is their proposed solution from the original paper[4]

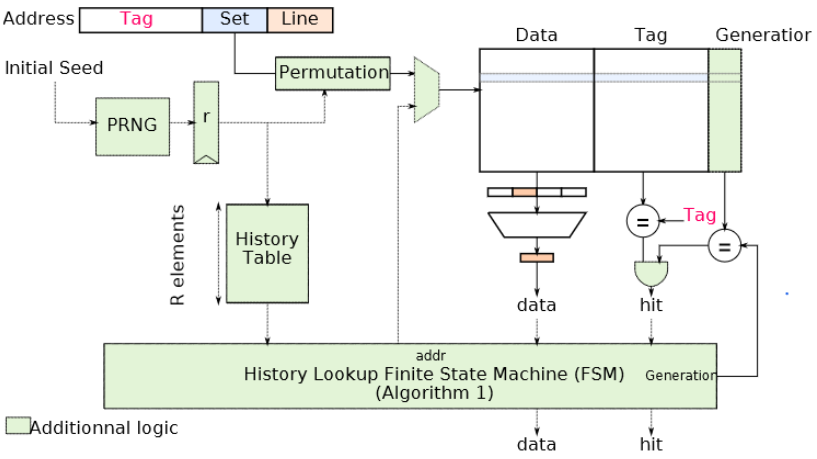


*Figure 1- chart taken from scramble cache paper [4] showing **their** the proposed solution.*

The "Scrambled Cache" scheme seems quite effective in simulations with very low miss penalty added by the mechanism and overall low performance penalty in software benchmarks.

What is missing in the "Scrambled cache" paper [4]*:*
- Only abstract emulation - no implementation.
- Abstract implementations using infinite counters for "Generation" (to keep track of the 'age' of each mapping- on what permutation iteration was the line brought to the cache)
- No "full picture" of cost -area, power, delay etc.
- No effectiveness demonstration – with simulating attacks.

What we are set on achieving with Our "Randomized Memory area solution" :
- Software and hardware implementation – usable by anyone!

- Cost analysis with a synthesizable processor
- Effective demonstration – ability to any C / C++ program with attacker/Victim simulation and benchmarks on an embedded CPU.

What is the Cswap - permutation?

To randomize the memory accesses we are using a permutation to determine the mapping between "original addresses " and where the data actually should be (not to be confused with virtual vs physical addresses - which in that case the mapping isn't random but arbitrary and won't change through the programs runtime and can be learned through reverse engineering [1]).
This permutation is calculated on the "Set" field of the address only to calculate the "New Set" -which is the randomized mapping.

Cswap permutation appears in [4] . The idea is to have a nonce called R - which is generated with a PRG. In our case, R is 20 bits long . Then, perform the following:

a. Xor the "Set" with bits [7:0] of R->xor res.
b. Now perform a series of "Cswaps" - conditional swaps on pairs of bits -on the xor result.
c. Start with pairs of bits at indexes which are 4 bits apart (0-4,1-5,2-6,3-7), then 2 bits apart (0-2,1-3,4-6,5-7), then 1 bit apart(0-1,2-3,4-5,6-7).
d. Each swap is conditional - and the bits are either swapped or not depending on the value of a third bit taken from the nonce -R . (for the first swap - bit 8 then 9,10 … all the way to bit 19)

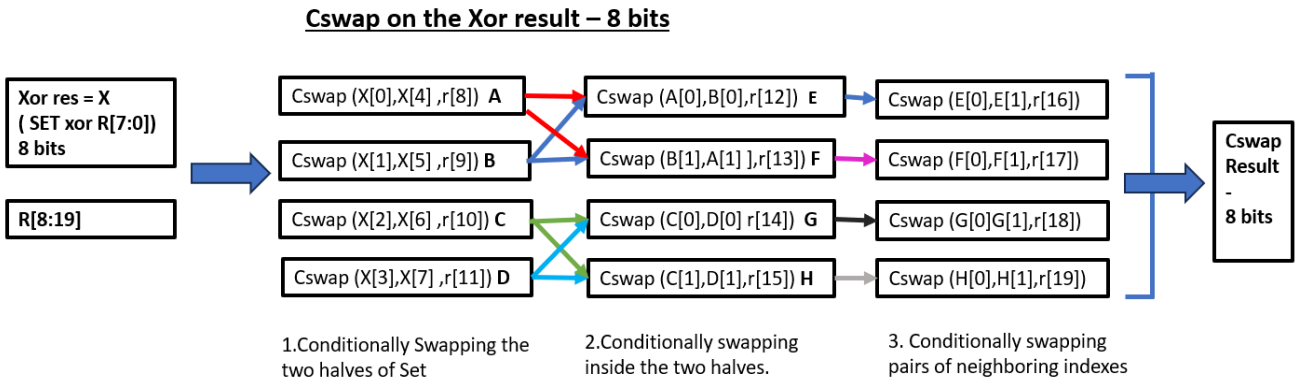


Figure 2-Schematic of the Cswap operation on an 8-bit set field (with 20-bit R/nonce value)

The characteristics of the Cswap permutation assure us that all bits in the xor result can end up in any bit index in the result of the permutation. Thus ensuring a significant effect for changes to any of the inputs of the permutation (input or secret key) and that the changes in input propagate well into the output.

Additionally, we have tested the permutation to be injective for all R values and starting addresses for the randomized memory area.
This ensures that two different sets will never map to the same place. (avoiding collisions)

3.The proposed approach

What parts do we need:

1. Finite - relatively small, **Sequential memory area (array)** -which the programmer initializes and then keeps using it to access the sensitive information stored there. The programmer should use the method/ machine instructions we have created.
2. **History table** -to keep track of changes in the mapping and to ensure concurrency of the data after the mapping randomly changes. Each entry in the table stores the "original set" which the data currently stored in the appropriate line in the memory area originally came from. This helps us "fix the mapping" - when needed.
   The history table can be kept in a register file to improve performance and possibly security in some cases (if the history table is very large- which is not the case for us).
3. **PRG -** generates **R** values. Possibly an additional history table for previous values of R for quick lookups.

The memory access flow will change to include permutation calculation to find the randomized location where the data should be according to the current mapping .
Let's look at a 32-bit address. The address is split to 3 parts - Tag, Set(index) and offset. For our examples we arbitrarily chose that the sizes of the fields will be :
4 bits for the offset (16 bytes in each block), 8 bits for the set (256 lines in cache) and the remaining 20 bits are for the "Tag".

Figure 3- hexadecimal address broken into its fields - a 4-bit offset, 8-bit set, and a 20-bit tag.
Now we apply the following algorithm whenever we are accessing the memory within the randomized memory area:

1. Extract the original "Set" field from the original address.
2. Calculate the "Cswap permutation" on the original set and get the "New Set" (using the current R value for mapping)
3. Then create the new address by replacing the original set in the address with the "New Set" from step 2.
4. Now, check the History table at the new Set (which represents the line index within the memory area in which the data should be according to the current mapping).
   If there is a match - this means that the data we want to read/write is at the correct location under the randomized mapping - and we can access that address - step 7.

   If not - continue to step 5.

5. Fix the mapping - search the history table for the line index that has the "original set" value .
6. After finding the match - fix the mapping by swapping the data of the line index from step 5 and the line at the "New Set" index - and update the history table accordingly.

7. Perform a memory read/write - access the memory at the “new Address”.

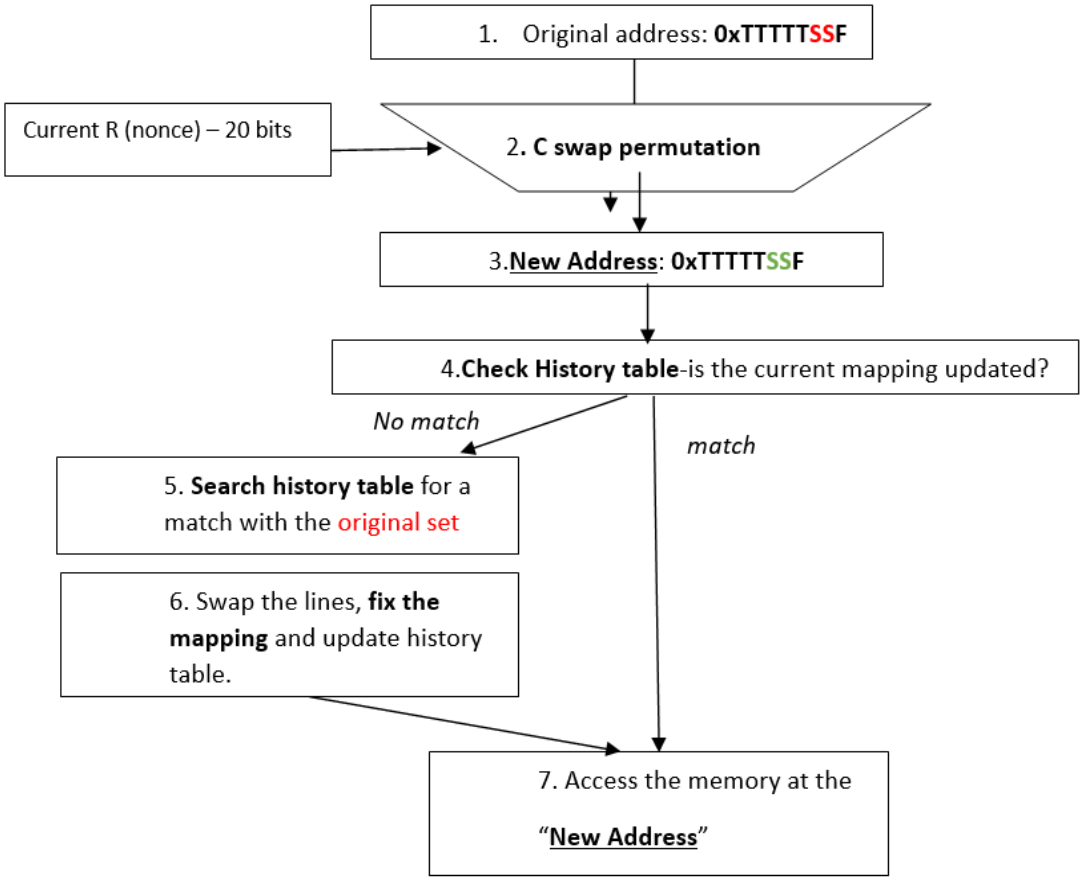


Figure 4 - a flow chart of the memory access flow.

In order to analyze our solution’s effect on the cache we need to understand when we have a memory access which affects the state of the cache - which the attacker process can monitor because the cache is a shared resource.
There is memory access only on stage 7 and this access will happen according to the randomized/permuted access pattern which can prevent the attacker from learning anything from time differences due to evictions. But if there is a mismatch with the history table on step 4 - we need to swap the contents of the lines. Which requires additional memory accesses at step 6.
We will later show, through statistical testing, that our solution is able to hide the SETs (the SET field of the address) access pattern of the Victim process from an attacker. [see figures 2-6]

4.The built solution summary

We created a C library with header files called “RandomizedMem.h”. (as well as additional headers for setup, initialization, various correctness and statistical tests etc.)

The C code is slower but allows us to have a working version of the scheme that can run on any computer that can have the C code compiled for it. As well as providing a reference and prototype for the hardware accelerated version which we created afterwards.

There are methods for allocating the memory area, setting it up (writing to it before applying the Randomized/permuted memory access) , defining the memory area, and finally, the memory access routines according to the memory access flow.

The Hardware Acceleration

We used tools made by Cadence which allowed us to work on an existing cpu architecture and add additional hardware instructions to the CPU build.
We used the “Cadence ® Tensilica ® Xtensa ® processors” -and wrote HDL implementation of the existing memory access flow methods from the C code. [6]
To quote from the TIE manual from cadence.com:
“Xtensa ® processors with new instructions and additional bandwidth using the Tensilica Instruction Extension (TIE) language enables you to compute and move data tens or hundreds of times faster than conventional processors”
“The advent of customizable processor IP changes the game. A much broader range of designers-not just processor designers can now develop processor IP that is tailored to specific tasks. These processors deliver the same or nearly the same level of performance as custom-built RTL blocks and offer several significant advantages.” [6]

We swapped all C functions with “Tie functions” [6] and then packaged the memory access flow into custom machine assembly instructions (“OP_PERMUTE_ADDRESS”) that are added to the processor’s **instruction set** .

The software IDE Cadance created (“Xplorer”) , allows for simulation of any c/c++ program on the custom CPU that was created. As well as providing compilation and cycle accurate profiling and debugging of the program running in simulation on the cpu.
We used these features to test the advantages of hardware acceleration and provide data on the cost of implementing our scheme in an embedded oriented CPU architecture.

5. Tests and measurements :

At first we tested **correctness** to ensure that the scheme simply works. This includes checking that the mapping is injective, sequential reading tests - bringing the memory area into a known state and then reading back from it - and checking that we are reading the same values written previously to the memory area. This has been checked to work before and after changing the permutation (by swapping the current R value) many times.

5. a : Statistical tests

Are we truly masking the information leaked by the sets that are accessed in the CPU's cache? (and possibly evicting the attacker's data or causing differences in execution time due to cache access pattern providing shorter runtime if there are more cache hits)

A way to ensure that will be to use statistical tests to show that the distribution of the sets that are actually being accessed - is uniform.

A well known statistical test is "Pearson's Chi-squared test" [7] that tests for the proportion (p) out of N samples. We would test the proportion against uniform distribution - 1/N . Which in our case will be: $1/(2^8) = 1/256$

First test : Running a N=15,000 sequential reads and counting how many times each set was accessed. We swap the permutation (R value) 10 times in equal intervals. We count for each SET (0 -255) how many times it was accessed.

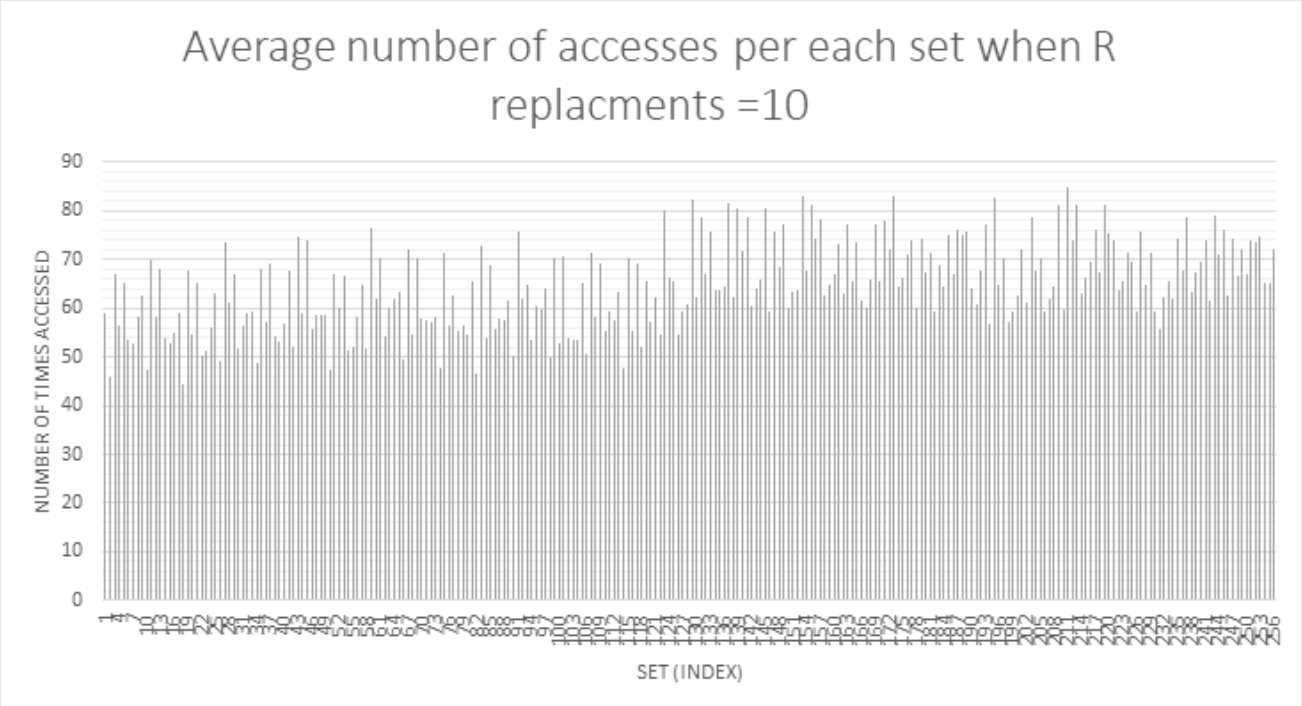


Figure 5 - average number accesses per each set for 15,000 accesses.

After using the Pearson's Chi-squared test we confirmed with high probability that the distribution is uniform across all 256 SETs.

Another question that was answered through the statistical analysis :
How – How frequently should we swap the permutation?
Swapping the permutation causes more mismatches (data is stored according to the old mapping and therefore when we try to access it we will have to swap blocks of data which introduces a significant time penalty to the memory access).
We cannot keep the same mapping /permutation for too long otherwise the mapping isn't random but arbitrary and fixed. Such mappings can be learned through reverse engineering and are leaking information because of the fixed correlation between "original set" - and the "new set" after the permutation.
We tested this in two ways : (again testing with N=15,000 sequential reads)

1. Test the SETS access patterns as we increase the number of times we swap the permutation (R value).

If we only use a single R value (one swap) - we can easily see a very recognizable pattern on the graph. This shows why many swaps are needed.

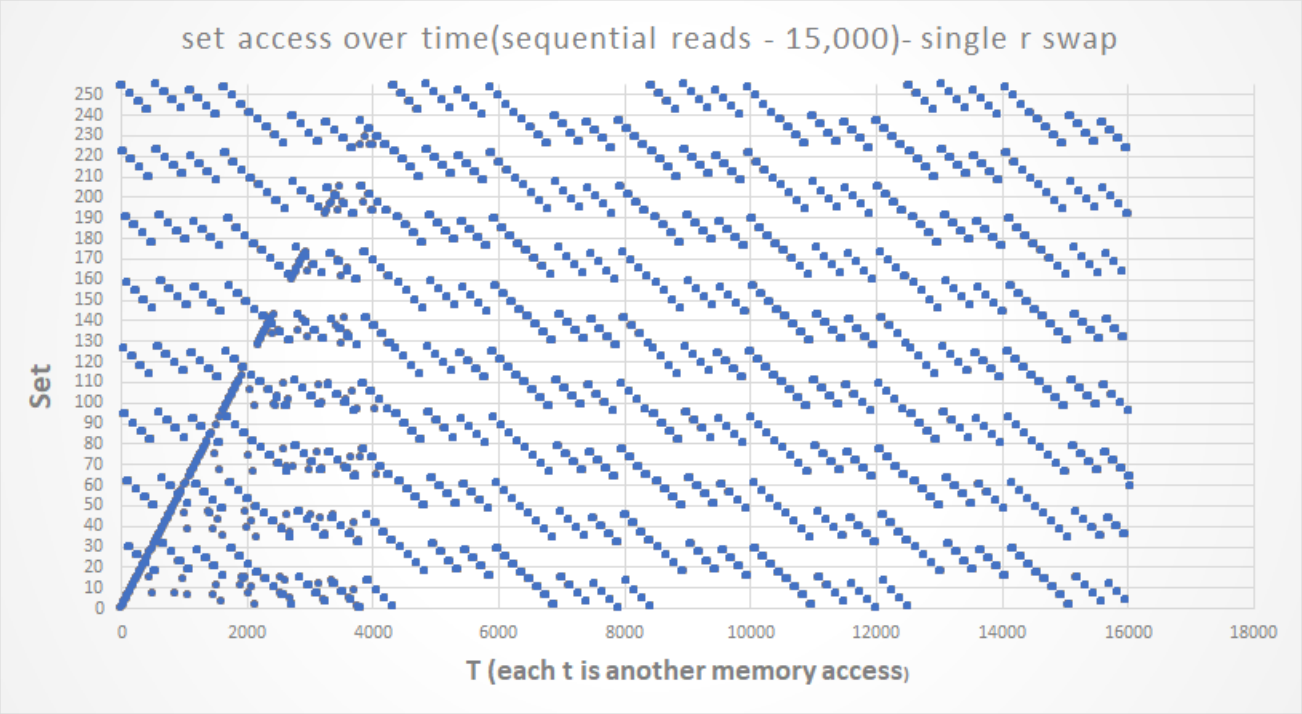


Figure 6 - set access over time - single R swap

We can also see in this graph a linear line at the beginning. This happens because at the start of the test we are swapping in almost every memory access as the memory area is not "organized " - the data is stored with the original mapping and as we access the memory the swaps are fixing the mapping to match the mapping according to the permutation (current R value) .

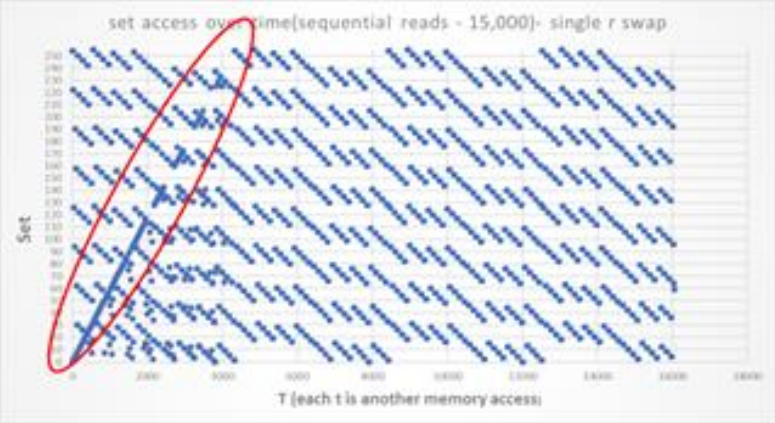


Figure 7 - set access over time - single R swap – the marked linear line shows the first shuffle of the memory

As we increase the frequency we can see that the pattern gets masked - and noisy - and we can no longer figure out which sets are being accessed.

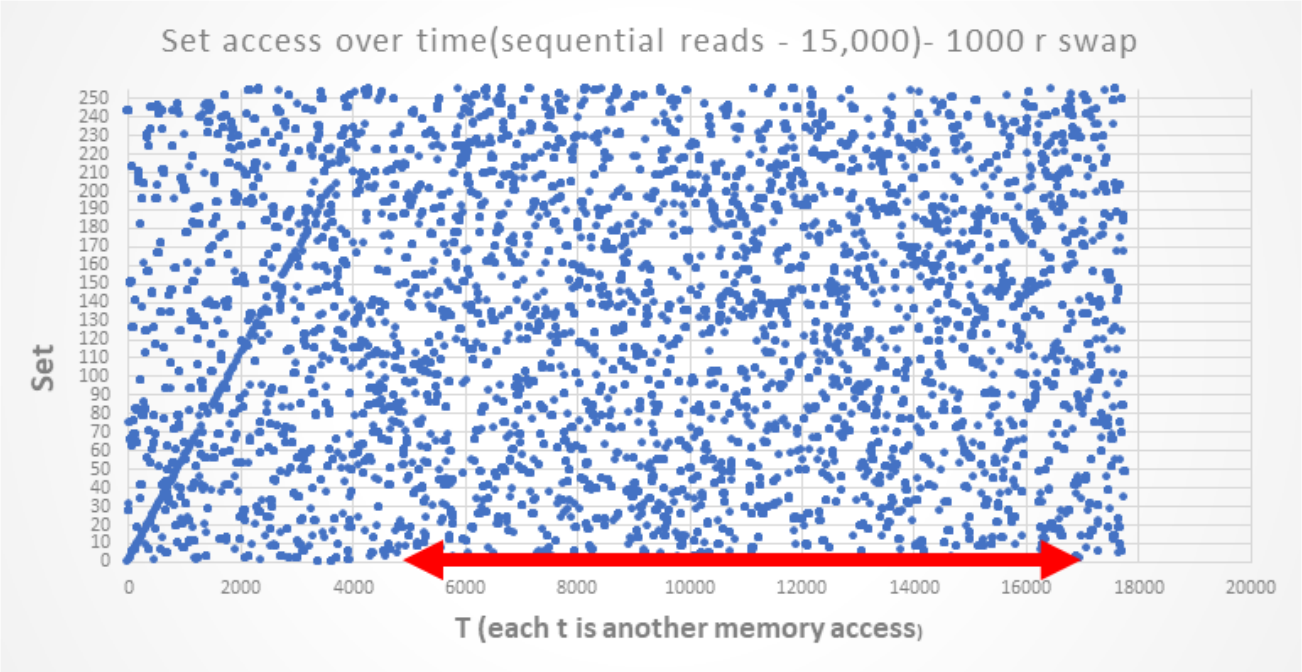


Figure 8 - set access over time - 1000 R swaps

And Finally:

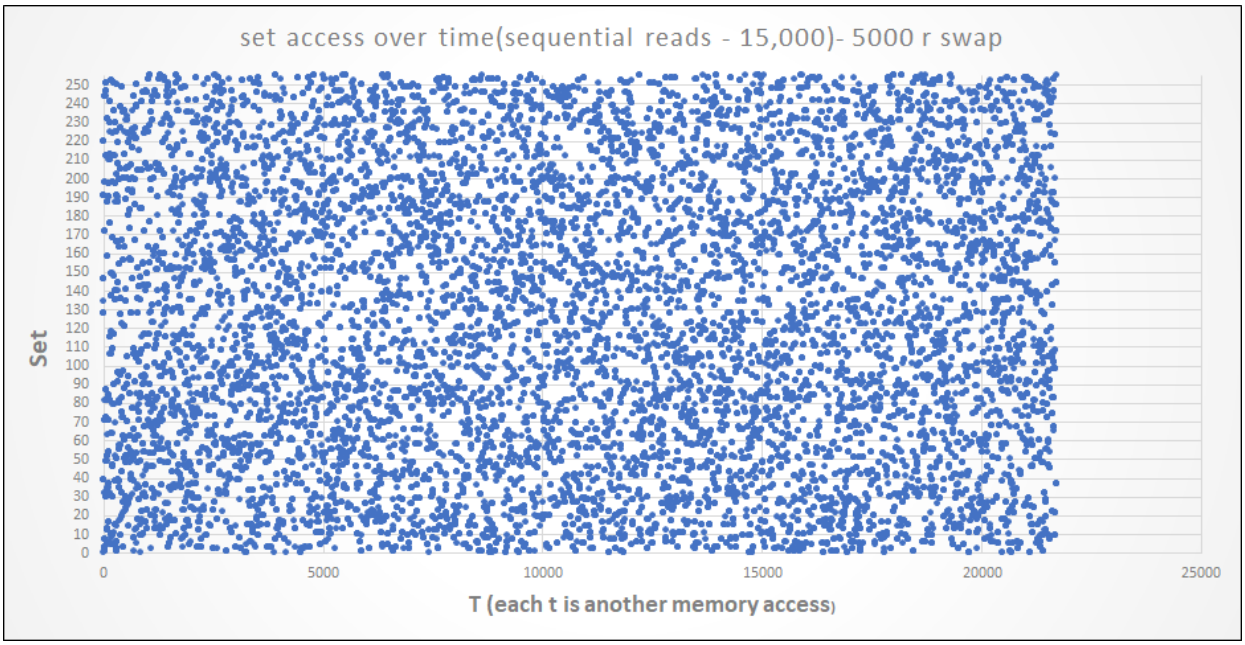


Figure 9 - set access over time - 5000 R swaps

2. Test the percentage of hit/miss on the **history table** (not the cache itself) to see in which frequencies we see a significant increase in the number of swaps we need to perform due to misses.

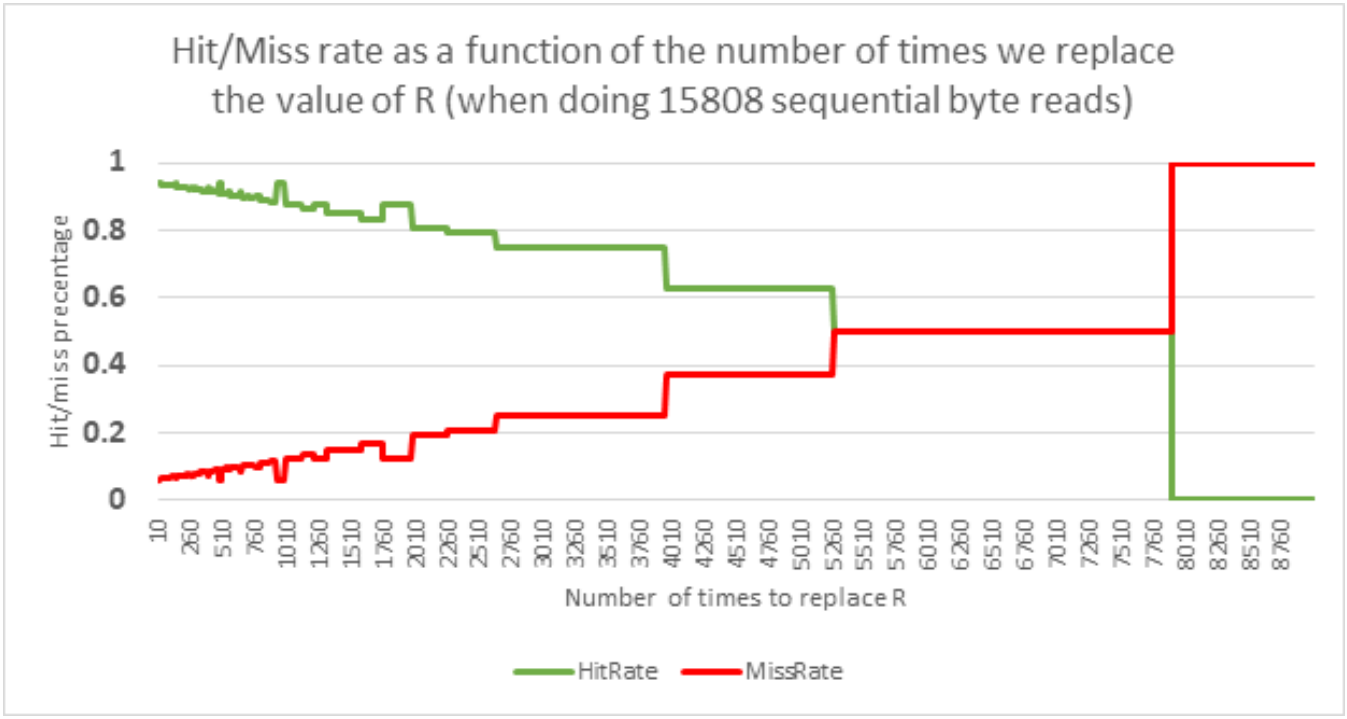


Figure 10 - Hit/Miss rate on the history table as the frequency of R value replacement increases.

3. 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 a function of the "number Of TimesTo Replace R" . We can finally answer the question :

What is the optimal frequency range to swap the permutation (swapping R)?

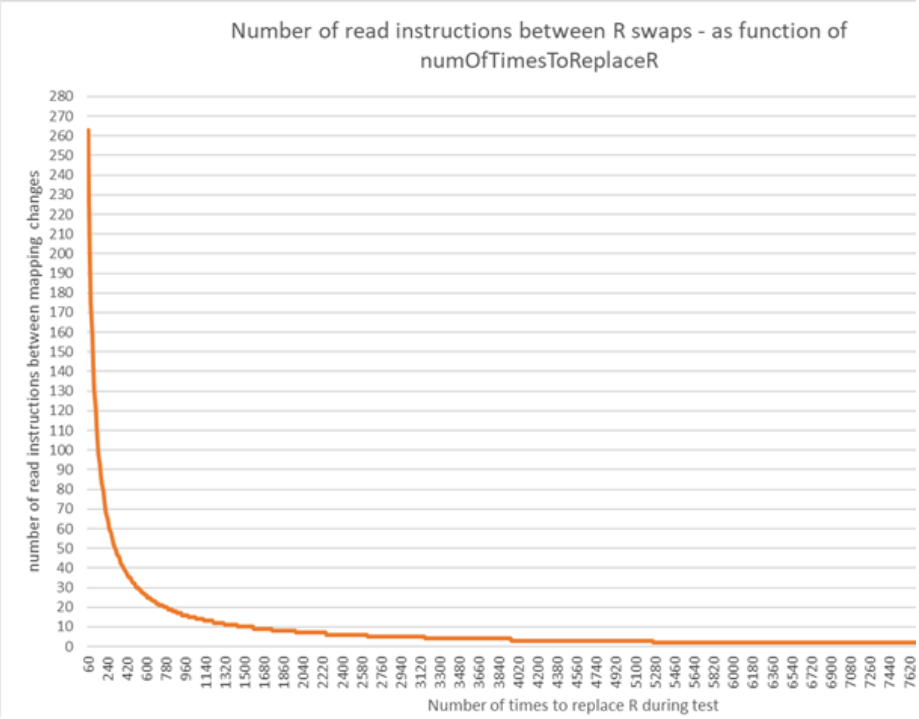


Figure 11 - the Number of read instructions between R swaps - as function of "numOfTimesToReplaceR"

The combination of this figure with figure 10 and 11 , tells us that the optimal frequency for swapping the R value is In the range of 900-1000 times in 15,000 memory accesses .Which means: R-value swap every 15-17 memory accesses.

Using this frequency, we get over 90% 'Hit' rate - The performance penalty at this swap rate.

And, as we have seen on figure 8 , which represents the 1000 swaps per 15,000 instruction case - we are able to successfully mask the information leaking from the cache SETs access pattern.

In conclusion, when implementing and activating this solution one could use our thumb rule of swapping every 15 accesses or swapping every time there is a context switch (the OS is switching between active processes). In that case, it is the responsibility of the developer/kernel programmer/hardware engineer to pick the frequency wisely while considering the tradeoff between performance penalty and security.

5.b. Performance test, penalty & acceleration benefits

Now we need to test the quality of the solution, the performance penalty from incorporating our solution and the performance gains from implementing the hardware acceleration.
We created and used benchmark programs. Firstly, we created a memory area and set its values into a known state. Then, we accessed the data using the three different methods – direct access (classic), using our program-only solution in C, and using the TIE solution. Then we compared the results. The benchmarks have been tested using the simulation and profiling tools with cycle accurate simulation using Cadence® Xtensa® Xplorer® IDE [6] .

The parameters that we are interested in measuring are:

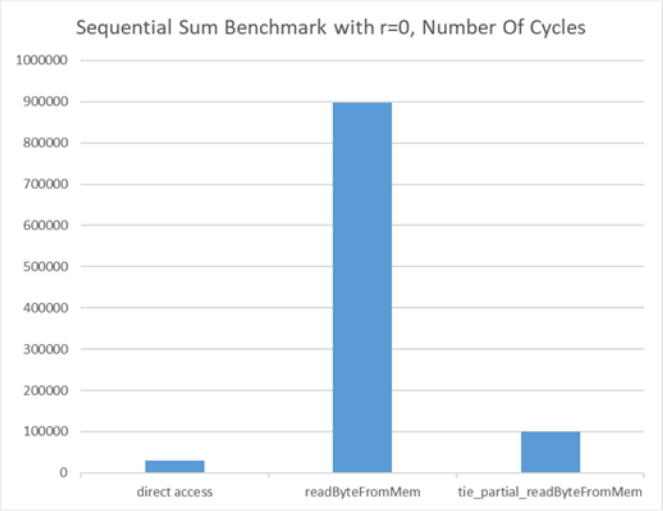
a. Number of cycles in which the processor runs the benchmark.
b. The run time of the benchmark program.
c. The ratio of hit-miss in cache.

The first benchmark is the "**Sequential Sum Benchmark**". The idea was to access the whole memory array sequentially, which means 4096 memory accesses as the size of the array. This tests the result we get if the programmer is trying to read data from the array in sequential manner - and therefore should benefit from spatial locality. For example: C style string operations, searching in an unsorted array etc.

The data in memory was set up using R=0. Then, we chose a random value of R to access the data. We did that for multiple values of R and averaged the results.

Note: "direct access" means that the permuted memory access is turned off, *"readByteFromMem"* is the C only version and "tie_partial_*readByteFromMem*" is the benchmark using the TIE costume instructions.

Results:
I. ***Cycle count***: for R=0 (same R value) was reduced by 89%.For R value that differs from 0 the average reduction was about **60%**.
II. ***Runtime***: for R=0 the speedup was 4.7 times, and for R=0 the average speedup is **2.3 times**.



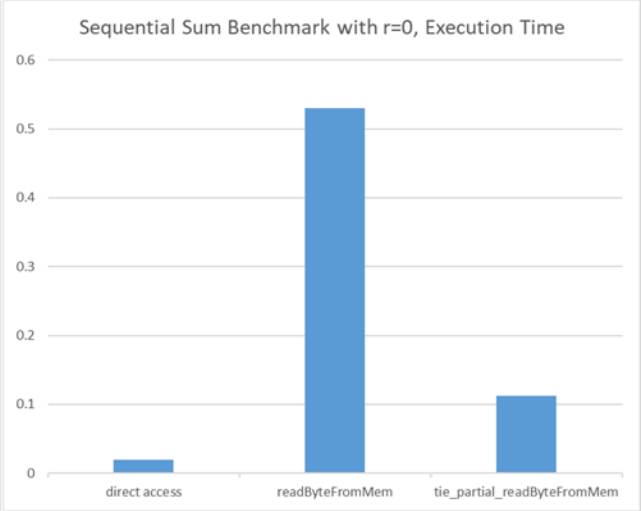


Figure 12- the runtimes of the sequential sum benchmark. Left: cycle count. Right: time in seconds.

The second benchmark is "**Binary Search Benchmark**". The idea was to implement a binary search in our memory array, which is a non-sequential benchmark due to an unknown data access pattern. This benchmark tests an access pattern that is not sequential and is closer to “Randomized access”. In this test the program won’t benefit from spatial locality.

The data in memory was set up using R=0, then we chose a random value of R to access the data. Due to the fact that our solution's efficiency is based on repeated data accesses, we ran the benchmark using the same values 100 times and averaged the results of each R & key values. We replaced the values of R & key and averaged the results.

Results:
I. ***Cycle count***: the average reduction in cycle count was about **72%**.
II. ***Runtime***: the average speedup is **2.7** times faster.

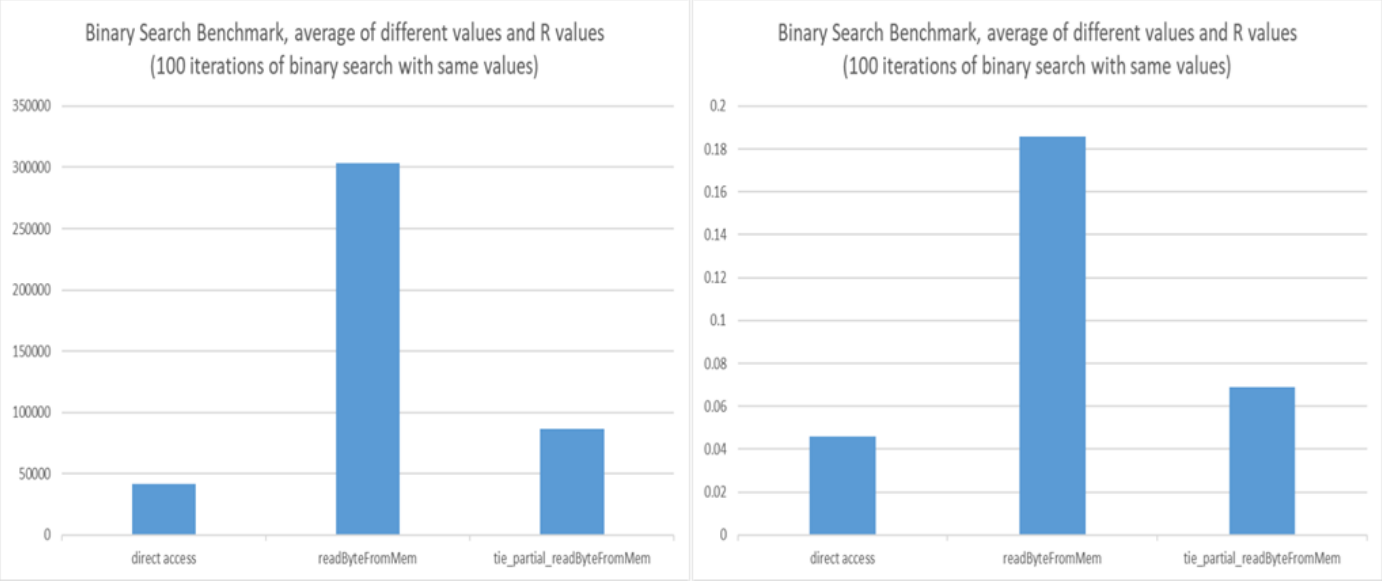


Figure 13 - the runtimes of the Binary search benchmark. Left: cycle count. Right: time in seconds.

6.Summary

As we learn about the risks that arise from cache timing attacks which utilize side channel leakage from the shared cache, We understand the importance of implementing solutions that can mitigate this vector of attacks. The threat exists for both high end server systems with complex and expensive CPUs as well as simpler embedded systems. That is why it is advised to consider relatively simple and 'cheap' to implement schemes that can protect embedded processors - both existing ones (through software) and newer ones - with the added hardware acceleration with custom logic to speed up the calculations and actions required for implementing our scheme.

We looked at previously offered solutions and created a scheme that can be implemented given the restrictions of real hardware and limitations of having to interface with existing IP that can't be changed easily such as the cache and more. We did use the effective and low cost "Cswap" from the Scramble cache paper [4] and the authors mentioned - it does provide us with a very cheap permutation than can be calculated in a single cycle.

Another takeaway from working on this project is the power of the customizable instruction set provided by the Cadence Xtensa architecture and suite of tools. These allowed us to implement our logic into an existing processor design, test, simulate, calculate costs in area and energy and in the end- get a synthesizable processor that can be manufactured. These tools significantly empower researchers that are looking to test their hardware acceleration ideas - and incorporate them into existing embedded oriented architecture.

7.Acknowledgments

The authors would like to thank Dr. Itamar Levi and Mr. Oren Gonen for their guidance and support throughout this project. Additionally, we would like to thank Dr. Levi for the hardware courses he taught which we took throughout our studies in the faculty -some during COVID times.

8.Sources

[1] Overview paper - Maria Mushtaq, Muhammad Asim Mukhtar, Vianney Lapotre, Muhammad Khurram Bhatti, and Guy Gogniat. 2020. Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA. Information Systems 92 (2020), 101524

[2] Overview paper and solutions - Ge, Q., Yarom, Y., Cock, D. *et al.* A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J Cryptogr Eng* **8**, 1–27 (2018). https://doi.org/10.1007/s13389-016-0141-6

[3] Overview solutions- Cache Attacks and Countermeasures: the Case of AES (Extended Version) revised 2005-11-20 , Dag Arne Osvik , Adi Shamir and Eran Tromer. dag.arne@osvik.no Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel {adi.shamir, eran.tromer}@weizmann.ac.il

[4] Scramble Cache paper : A. Jaamoum, T. Hiscock and G. D. Natale, "Scramble Cache: An Efficient Cache Architecture for Randomized Set Permutation," 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 2021, pp. 621-626, doi: 10.23919/DATE51398.2021.9473919.

[5] Meltdown paper - Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M., & Strackx, R. (2020). Meltdown. *Communications of the ACM*, *63*(6), 46–56. https://doi.org/10.1145/3357033

[6]https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/ip/tensilica-ip/tip-tie-wp.pdf

[7] Karl Pearson and the Chi-Squared Test , R. L. Plackett
International Statistical Review / Revue Internationale de Statistique
Vol. 51, No. 1 (Apr., 1983), pp. 59-72 (14 pages)

[8] Another overview with pl and rp cache– mainly AES : Kong, J., Aciicmez, O., Seifert, J.-P., & Huiyang Zhou. (2009, February). Hardware-software integrated approaches to defend against software cache-based side channel attacks. *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. http://dx.doi.org/10.1109/hpca.2009.4798277