**Side Channel Attacks Analysis for Secure ISA and Architecture**

By:

Tanushree Tanushree ([tanushre@usc.edu](mailto:tanushre@usc.edu))

Richmond Johnson David Bhaskaran ([davidbha@usc.edu](mailto:davidbha@usc.edu))

Abhishek Barve ([abarve@usc.edu](mailto:abarve@usc.edu))

**Problem description:**

The goal of the project is to determine the presence of cache side channel attacks on a Multi-core processor design. The goal of the system is to analyze certain characteristics of the systems like Cache latencies and access pattern to determine the behavior for cache side channel attacks.

The objective is to analyze the presence of a cache side channel attacker and if the attacker can access cache lines of the optimized Multi-core processor. Then, we are planning to add cache security as a parameter to analyze the performance of baseline design and design after NoC optimization using novel algorithm.

We have decided that the attacker will be present as an application program which has infiltrated into the system and is now running on one of the cores of the design.

We are modelling a cache hierarchy model with L1 and L2 cache for 3-core system in which 2 cores will run the victim programs and 1 core wil vvvvv l run the attacker program which will issue load commands in round-robin algorithm for specified cache blocks on shared L2 to check the if that cache block was already present in L2 due to victim program.

**Project timeline:**

By phase I, we studied about different cache attacks and created a plan to demonstrate the same on a multi-core, multi-level cache hierarchy system. In order to implement this, we gathered the required tools and sketched out a rough layout of the plan. We mainly divided the project into 3 segments for equal workload distribution. Segment 1 (undertaken by Richmond) deals with understanding of the tool that provides us with assembly code which we plan to simulate. Segment 2 (undertaken by Tanushree) deals with modeling of multi-core, multi-level cache hierarchy system based on the output assembly and demonstrate basic cache attack. Segment 3 (undertaken by Abhishek) deals with implementation and analysis of novel cache attack prevention measures.

During phase II, a basic outline was drawn for the output assembly code and based on that, the cache was modelled successfully. A basic victim and attacker trace (consist of series of loads/stores) was generated using a python script in the desired format (similar to LLVM tool). After analyzing the results, we were able to deduce that the attacker is successfully able to identify victim cache blocks.

Finally, in phase III we will be able to combine the tool’s output with the cache model and ran the integrated system to produce the desired results which successfully simulate the cache attack. Also, for attack prevention, we have come up with a novel technique where we plan to populate the victim trace with arbitrary load/stores to distract the attacker. Though it can tend to decrease performance for certain benchmarks, but it surely is successful in reducing the attack percentage (which is the main goal of this project).

Further improvements can be thought in the direction of smart population of victim trace such that it does not hurt performance and reduce the attack ratio.

**Side Channel Attacks Analysis for Secure ISA and Architecture**

Tanushree Tanushree, Richmond Johnson David Bhaskaran, Abhishek Barve

Department of Electrical and Computer Engineering,

University of Southern California

Email id: {tanushre, davidbha, abarve}@usc.edu

***Abstract –* With increasing micro-architectural attacks in modern day processors, it has become necessary to secure the programs. One of the possible attacks that needs to be reduced is Side Channel Attack on Shared Cache for a multi-core processor architecture. There are various attack methods that demonstrate how an attacker can access a victim programs’ content using shared cache.**

**In this report, we will explain the implementation of cache attack on a multi-core, multi-level cache hierarchy. We will discuss mainly 3 different types of cache attacks namely “Evict + Time”, “Prime + Probe” and “Flush + Reload”. After successful demonstration of cache attack, we will use a novel attack prevention technique to reduce the attack ratio and finally prove that this technique can be used to reduce attack without much overhead.**

1. INTRODUCTION

Today’s modern processors comprise of numerous micro-architectural components, out of which cache is one of the most extensively used block. Cache is a smaller SRAM structure that holds the most recently used data. It’s smaller but faster as compared to RAM. Rather than creating only 1 bigger cache, we create a hierarchy of caches (L1, L2 and so on) to better utilize the memory accesses and reduce the latency. Due to need for more parallelism, the processors now have multiple cores with each core running an independent thread of program. Also, each core has its own private L1 cache. However, to synchronize all cores, the lower level cache L2 serves as a common ground and hence also known as shared cache.

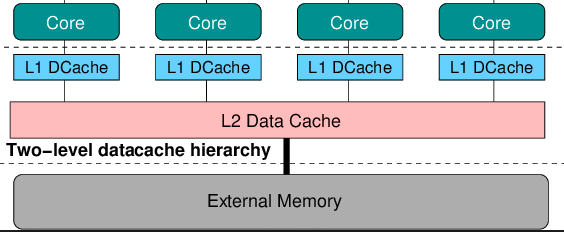


Fig 1. Cache Hierarchy

This concept of shared cache exposes a lot of cache attack possibilities. Cache attack in this context means identification of cache blocks being accessed by a program to an attacker program. If attacker program can access the blocks used by victim program, it can modify that location with some spurious data.

There are mainly 3 big cache attack methods, namely “Evict + Time”, “Prime + Probe” and “Flush + Reload”. In Evict + Time, the victim program is executed and timed; then a cache block is randomly evicted from L2; then again victim is run and timed. If the time taken for a block is less this time, then victim probably accessed that block. In Prime + Probe, shared cache is populated with cache blocks, then evicted and again populated. If the time taken remains same, the probably it was accessed by a victim program. In Flush + Reload, attacker flushes the shared cache and waits for victim to complete and then again reloads the cache[3]. If the reload time is less, then it means victim used that block. These attacks are discussed in detail in the section II.

As for attack prevention, a novel technique is used where the attacker is distracted by populating the stream of victim memory accesses with extra loads/stores. This way the attacker prediction is violated and hence the program becomes more secure. Further details are present in the section II. Finally, in section III, observations and results are presented and room for further improvement.

1. FRAMEWORK

In this section we will describe the overall framework of our cache attack and prevention simulation. This section is divided into 3 sub-sections. First section deals with explanation of LLVM compiler tool, with the help of which we are able generate assembly code for our victim program and using that assembly code we extract a trace of memory operations. In the second section, we discuss how a multi-core, multi-level cache hierarchy is implemented in C++. An attacker trace (generated by python script) will be run along side the victim trace will be run on this cache simulator program which will generate a log file that lists out all the blocks used by attacker with their latencies from before and after the attack thereby enabling us to identify the cache blocks that were accessed by the victim. In the third section, we will discuss cache attack prevention strategies and how effective they could prove useful in real world scenarios.

1. ***LLVM Toolchain, Clang Compiler and Address Space Randomization***

The tool to be presented as a result of this approach will be an llvm-based tool that takes in .ll files as input and performs manipulation on the files that are generated henceforth in order to achieve the address randomization. LLVM is an umbrella toolchain that incorporates support for several languages in the front-end such as C, C++, Fortran, Haskell, JAVA, Ruby etc. while supporting several hardware backend architectures like x86, amd64, PowerPC, SPARC, NVIDIA etc. This is achieved using LLVM’s powerful tools like the Clang compiler and a number of optimizers/linkers etc. Given any input language, LLVM converts them into an intermediate IR-representation. IR is a strongly typed RISC-like Instruction Set Language. With this, LLVM performs several optimizations such as Compile Time Optimization, Link Time optimization etc.

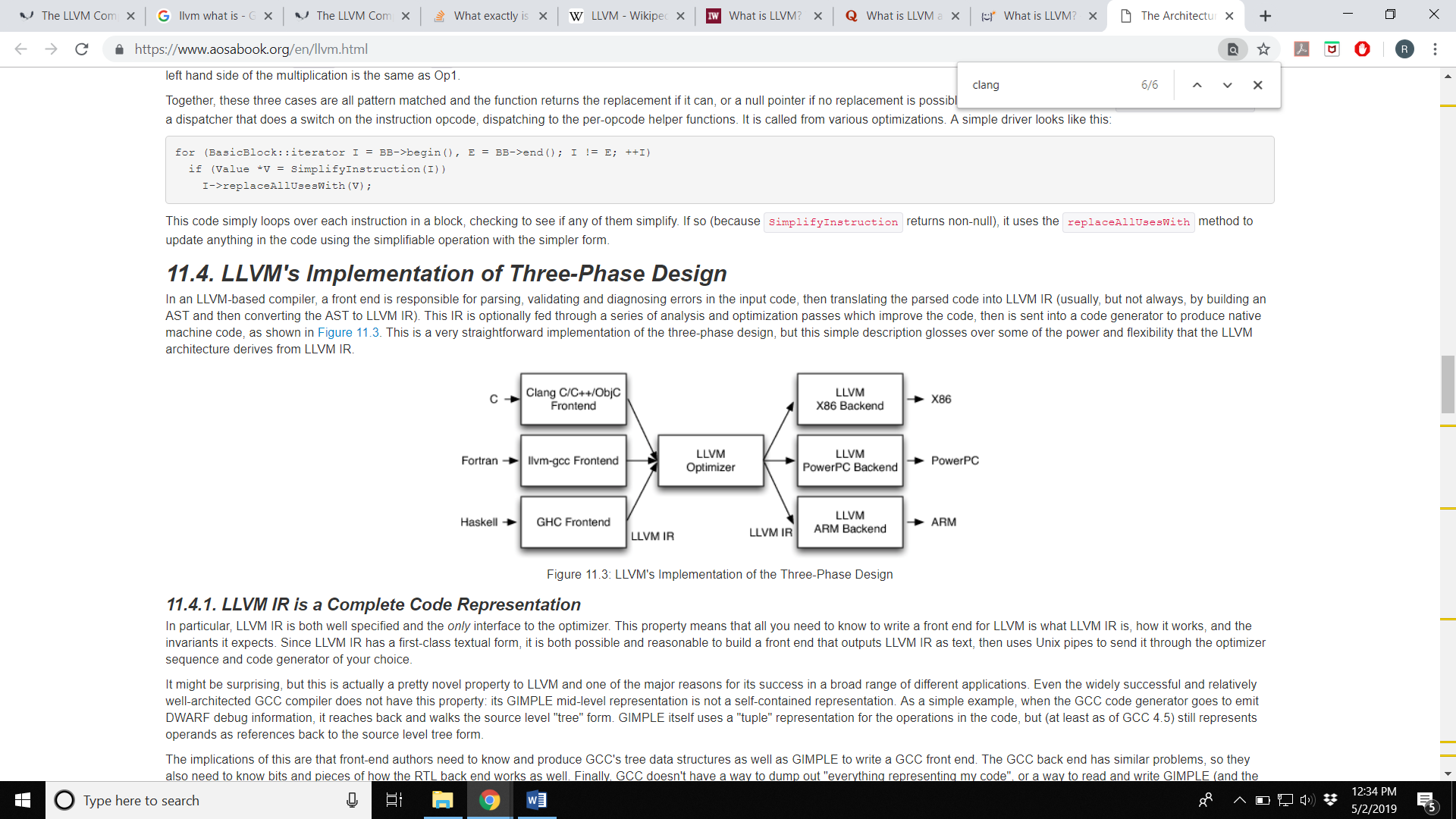


Fig 2: LLVM Infrastructure [5]

In the tool presented as the final output, the intermediate IR files generated in the .ll format is intercepted and converted into assembly files (.s format). The address space is further analyzed to look for critical variables. By the theory of address space randomization, including more variables within a stack frame and performing a number of arithmetic/logical operations on them, without considerable loss in performance is a basis to confuse any outside attacker from knowing where the critical data lies. This is probably in line with the pseudo random noise sequence generators in communication theory, wherein by making a program and its memory footprint reasonably large, a potential attacker will always get the impression that the program is in fact dealing with a number of variables. These changes in the address space is made feasible by a careful study of the assembly files and making sufficient modifications without disturbing the functionality and at the same time, not compromising the stack spaces. In the end, a graphical representation [4] of the effectiveness with and without the tool will be given to better comprehend the results. By operating on the LLVM level, it is ensured that the tool will be able to support any front-end language, just by the sole fact that the LLVM in turn supports multiple languages. The assembly file analyses are performed on the at&t format. The tool can be invoked in the final stages of the LLVM linking and optimization.

1. ***Cache Simulator and Attack***

In this sub-section we will first discuss how we designed a multi-core, multi-level cache hierarchy system in C++. Then we will focus towards running victim and attacker traces obtained from the compiler on this simulator to demonstrate a successful cache attack.

Since the simulator had to be developed to model multiple caches, we opted to use C++ and its object-oriented concepts to employ re-usability while creating multiple structures. The first step in building a cache model was to write a basic “Tag\_Block” structure that consists of following information:

1. Valid – Meaning the cache block is valid
2. Dirty – Meaning whether the block is modified
3. Lru\_counter – A “least recently used” counter which indicates the temporal access of a block
4. Tag – tag information of that block

Also, to implement MSI coherence protocol, another class called “Coherence\_info” is created which is inherited by the Tag\_Block class. The members of this class have been kept private with their respective access APIs as set/get functions.

Multiple objects of this class are created inside another class called “Cache” which has all the functionalities of a regular cache. Based on the cache structure (discussed later), an array of Tag\_Block objects are created which represent the tag array part of cache (data array not required as we only deal with accesses and not the actual data). It contains the following parameters:

1. Global\_lru\_counter – Used to keep a count of recently accessed blocks
2. Miss – counts the number of misses
3. Hit – counts the number of hits
4. My\_blocksize – Blocksize used for the cache
5. My\_assoc – Associativity used for the cache
6. My\_sets – Number of sets in the cache

The constructor is overridden to create either L1 or an L2 cache using the same class. Simply passing the level (1: for L1, 2: for L2) as an argument while creating a cache object can create the desired cache.

In order to explain the Cache class functions, we need to understand the working of a single level and multi-level cache. Let’s say a processor sends out a read request, it goes to the cache control unit which searches for that address in its cache. If it is not found, then the request is sent to the lower level (either L2 or memory). If that cache block is available in the lower level, it is fetched and placed in the cache (making it the most recently used block) and the required data is sent back to the processor. If it is found in cache, data is sent back to the processor and that block is made the most recently accessed block. For a write request, similar lookup takes place, but now the write data is written into the cache and that block is marked dirty, as well as, most recently accessed. If the cache block is not present and the set is filled with existing blocks, then while fetching the block from lower level, it replaces the least recently used block (using lru counter information) and places it in that location. The replaced block is either sent to the lower level or discarded based on its dirty bit setting (0: can be discarded, 1: need to be written back to lower level). This is called a write-back write-allocate cache where the dirty evicted block is written back to the lower level cache, as well as, upon a cache miss, a block is always placed in L1, even though it was accessed only once[2].

For a multi-level system, if a block is found in L1, it simply returns back/write the data in that cache. Whereas, if there is a miss, the request is sent down to L2 cache which searches for that block in its own cache. If found, it sends that block to L1, otherwise it sends the request further down. Evicted dirty blocks are placed in the lower level caches[2].

Now, in a multi-core system, each core has its private L1 cache, while L2 is shared between all the cores in a cluster. In order to maintain synchronization between the two cores, we use coherence protocols (we used MSI protocol for convenience). Coherence refers to maintaining the same latest data for a block across all the individual private L1 caches[2]. Following is the state diagram for MSI:

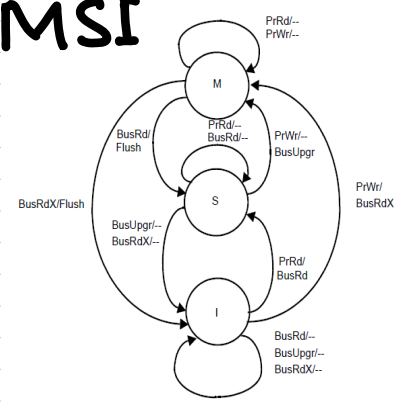


Fig 3: MSI Coherence Protocol

According to MSI protocol, if a block is not present in a cache, it is marked as Invalid (I state). If it is present in the cache and has not been modified yet, it resides in Shared (S state). And if it is modified, it is present in Modified (M state). When a processor requests to modify the block, it sends a request on the bus to invalidate that block from all the other private L1 caches (if any). This way, if another L1 requests for that data, the L1 having this block in M state will deprecate to S state and flush the latest data to L2 and provide it to the requesting L1 (which will go in shared state)[2].

As per our simulator, our primary function inside Cache class is “cache\_install”. This function performs search in the cache for the requested block. If it is present, it updates the lru\_counter and dirty bit based on the request (read/write). If it is not present, it first searches for an empty location in the relevant set and tries to place the block there. If no empty location is found, the least recently used block in that set is taken out and the incoming block is placed there. Finally, it sends the information about evicted block.

There are auxiliary functions in Cache class to perform repetitive tasks, such as function “cache\_search” returns whether the requested block is present in cache or not. Following is the list of functions present in this class:

1. get\_blockbits - find block bits from a given address
2. get\_setbits - find set bits from a given address
3. get\_tagbits - find tag bits from a given address
4. cache\_search - search a given tag in a cache by indexing using the set and block bits
5. search\_in\_set - find an empty block using valid bit in a given set
6. search\_in\_full\_set - Find the least recently used block in a given full set
7. address\_reconstruct - reconstruct the address using set, block and tag bits which is used by L2 cache when a block is eveicted from L1 cache
8. clflush – to invalidate the entire cache (making valid field for every block=0)

In the main program of our simulator, we create 3 caches (two L1 caches and one L2 cache) as per our system. On one L1, we will simulate our victim trace, while on the other L1 attacker trace is simulated. For this project, we have assumed arbitrary L2 hit/miss latencies.

Following are the cache configurations used for this simulator:

|  |  |  |  |
| --- | --- | --- | --- |
| **Cache** | Block size | Cache size | Associativity |
| L1 (both) | 32B | 8KB | 4 |
| L2 | 32B | 128KB | 8 |

Table 1: Cache Configurations

Now, let us demonstrate the cache attack. Among the 3 mentioned attack methods, we have chosen to implement Prime + Probe method as it is fairly accurate and gives us a live analysis of the sets accessed by the cache. In other methods such as Evict + Time, we do have to wait for all the programs to finish and we do not receive live update. Whereas in Flush + Reload, it requires shared memory which increases the implementation overhead (though being the most accurate of the 3)[3].

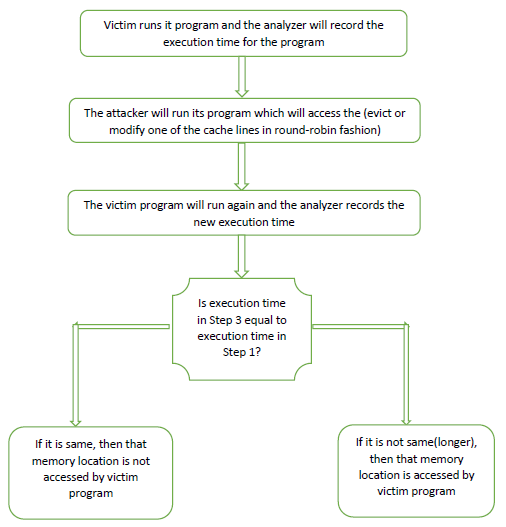
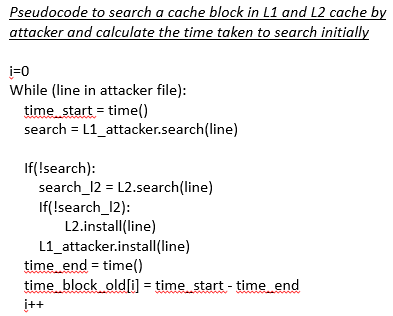


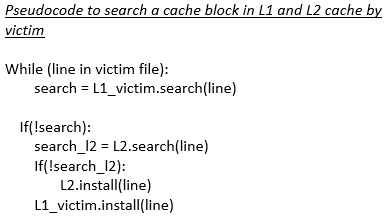
Fig 4: Cache Attack Algorithm

Our focus is to demonstrate cache attack prevention, hence Evict + Time is the most flexible as well as precise attack method. An attacker trace is generated by a python script which contains a series of loads to various cache blocks that fills up L2.

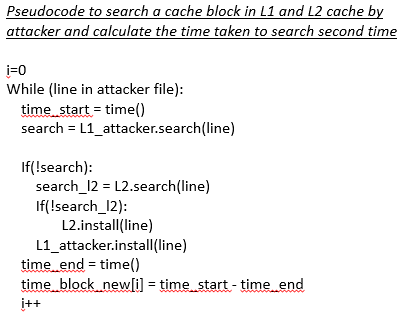
After creating all 3 caches, we first run the attacker program on an L1 (designated to attacker program). Upon its completion, we note down the time accessed by all the blocks. At the start L2 is empty, hence all the blocks would result in L2 miss and take a lot of time. Next, we run the victim program which accesses its own blocks. There maybe some blocks which are common between attacker and victim. Those block requests from victim will hit in L2, while others will miss. After the completion of victim, the same attacker program is re-run and time is again noted down for all the blocks. This time, if there is a difference between old and new times for a block, we can say that it was accessed by victim. If it had not been accessed by victim, it would have gotten evicted and upon re-accessing by attacker would have resulted in the same (higher latency) time as before. Since it was accessed by the victim, it stays in L2 and upon re-accessing, it returns with lower latency. This latency information is printed out in the excel file along with block ID and address.



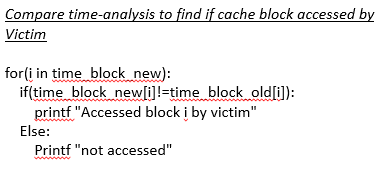
Code 1: Pseudocode for attacker program running and noting down the run time for 1st time



Code2: Pseudocode for victim program



Code 3: Pseudocode for attacker program running and noting down the run time for 2nd time



Code 4: Pseudocode for time comparison between 1st and 2nd runs

1. ***Cache Attack Prevention***

Write something (to be filled by abhishek)

1. RESULTS

To be written in the end

1. REFERENCES

[1]<https://conference.hitb.org/hitbsecconf2016ams/materials/D2T1%20-%20Anders%20Fogh%20-%20Cache%20Side%20Channel%20Attacks.pdf>

[2] EE457 Class notes: <http://www-classes.usc.edu/engr/ee-s/457/EE457_Classnotes/EE457_Chapter9/EE457_MSI_MOESI_Cache_Coherency_Protocols.pdf>

[3] <https://www.blackhat.com/docs/asia-17/materials/asia-17-Irazoqui-Cache-Side-Channel-Attack-Exploitability-And-Countermeasures.pdf>

[4] <https://peltiertech.com/open-csv-file-chart-csv-data/>

[5] <https://www.aosabook.org/en/llvm.html>