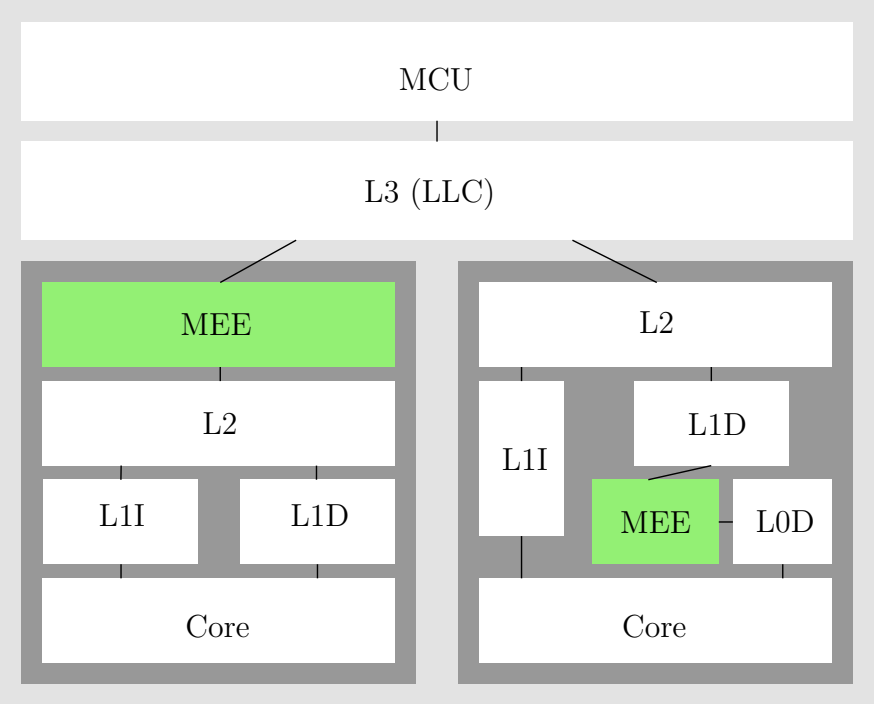
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Deep Integration of a Memory Encryption Engine in Modern Processor Designs

Master's thesis in Embedded Electronic System Design

IVAR SÖRQVIST

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Deep Integration of a Memory Encryption Engine in Modern Processor Designs

IVAR SÖRQVIST

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Deep integration of a memory encryption engine in modern processor designs

IVAR SÖRQVIST

Supervisor: Per Stenström, Dep. of Comp. Sc. and Eng.
Examiner: Per Larsson-Edefors, Dep. of Comp. Sc. and Eng.

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

Deep integration of a memory encryption engine in modern processor designs

IVAR SÖRQVIST
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Keeping execution of data secure from potential attackers is a major concern today, especially in cloud computing. Intel SGX is one example of such a trusted execution environment, utilising isolation of data on-chip memory and encryption off-chip. However, numerous publications have been published exploiting its vulnerabilities with different types of side-channel, Spectre and Meltdown attacks. In this thesis, we propose a relocation of the encryption stage deeper into a processor's memory hierarchy which could be a potential solution for a more secure system. We introduce two systems: first moving the encryption stage to before the shared last-level cache, second encrypting before the first level data cache with an added dedicated cache for cryptography. For the second different placements of prefetching are investigated further. Through simulations using the gem5 simulator, we show that these systems suffer minor performance losses compared to using no encryption at all.

Keywords: Master, thesis, processor, cryptography, encryption, engine, MEE, simulation, gem5, TEE.

Acknowledgements

I would like to express my deepest appreciation to Per Stenström that have supervised me in my work. Your invaluable expertise, advice and belief in my abilities have been fundamental pillars in writing this thesis.

Ivar Sörqvist, Gothenburg, June 2021

Contents

List of Figures

1

Introduction

With constantly increasing connectivity, more data are processed and communicated now than ever before. Amidst this trend, security concerns cannot be neglected since being able to keep your data safe and secret from potential attackers is a high priority among vendors. To achieve this, not only is it needed to encrypt our data when communicating with external sources. It is also a good idea to protect the data from malicious software with access to our hardware. In modern General Purpose Processors (GPP), a common approach is to separate secure processes from all others. One way to do this is with the use of Virtual Machines (VM) which there has been considerable research exploiting its weaknesses [1, 2, 3, 4]. A VM can not be fully secure, among others due to its submissive relationship to the system's hypervisor. Another way of keeping executing data secure is with the use of so-called Trusted Execution Environments (TEE), offering greater security and performance compared to VMs, which is made possible by relying more on customised hardware [5, 6, 7]. An example of this is Intel's Soft Guard eXtension (SGX), enabling secure enclaves for a thread to operate within and where all privileged software around is assumed to be malicious [8, 9]. By separating the secure process into a one-way transparent enclave the secure process can access resources from other processes up to the same privileged level, while processes located outside the enclave cannot access resources residing in it. Similar approaches to a TEE are available from AMD, named Secure Encrypted Virtualisation (SEV) and Arm, named TrustZone [10, 11]. In SGX and SEV, data belonging to the secure process is encrypted in the processor's main memory. It is kept encrypted using a Memory Encryption Engine (MEE) placed in the Memory Control Unit (MCU), which is then located between main memory and the Last Level Cache (LLC) and can be decrypted back to LLC when needed [12, 9, 10].

Even though these TEE's are offering a great deal of security, they have multiple times been proven vulnerable and that an attacker can access sensitive information [13, 14, 15]. Notably, Intel's SGX has had more attacks published than the others. However similar attacks seem applicable to AMD's and Arm's versions as well. Different cache-based attacks such as the CacheZoom, using Prime+Probe attack techniques, extracting encryption keys by accurately measuring cache access times when the encryption is done within an SGX enclave [16, 17, 18, 19]. Another interesting attack is the Foreshadow attack, based on the same speculative behaviour as the Spectre and Meltdown attacks, it can leak enclave secrets in plain

text [20, 21, 22]. Some attacks have even shown to be successful in placing malicious code within the enclave itself [23].

Henceforth, the security level offered in today's TEEs can be questioned. One possible solution to increase the security offered could be to integrate the MEE deeper into the on-chip memory of the GPP. We will focus on the performance of such systems and only lightly discuss if this would solve today's security issues. With that said, this thesis aims to:

- Investigate performance difference of systems with encrypted data integrated deeply into the on-chip cache memory. Compare this to systems representing today's TEEs and no encrypted data at all.
- Examine the system's sensitivity of different encryption/decryption overheads of the MEE.
- Explore the possibility of adding a dedicated cache, only accessible by a single thread, for encrypted data.
- Apply performance-enhancing techniques, such as prefetching.

To quantify the aim of this thesis, models representing the different systems are simulated using the gem5 detailed architecture [24]. This model uses the x86-64 architecture and is inspired by Intel's Skylake server architecture [25]. Moreover, the simulations are done using the SPEC cpu2006 benchmarks measuring times of 10 million instructions with warmed up caches [26].

From the simulations conducted it could be seen that moving the MEE down to before the LLC gave only minor performance losses. Moving it even further down to before the Level 1 Data (L1D) cache gave large performance loss but this could be decreased drastically with adding a dedicated Level 0 Data (L0D) cache and even more by using prefetching.

This thesis first goes through valuable background concerning modern computer architecture, TEE, MEE and some attacks targeting discussed TEEs. The proposed systems and how these were simulated is then laid out. Lastly, results from simulations and a discussion are provided.

2

Background

In this chapter, we describe the necessary background and theory for the thesis. Starting with a description of a GPP, then moving on to a description of TEE, MEE and finally a description of attacks that these TEEs have recently faced.

2.1 General Purpose Processor

When talking about microprocessors today, most people refer to the GPP which embrace nearly all Instruction Set Architectures (ISA) [27]. These GPPs are powering everything from our personal computers, smartphones to servers and supercomputers. What makes them general, is the design that lets them handle multiple different tasks. To define on a low granularity level what the GPP can do, the ISA is used. With it, the programmer can see in detail what the GPP is hardwired to do, with instructions such as WRITE, READ, ADD, and COMPARE etc. usually referred to as assembler instructions. By combining these low-level instructions, more complex tasks can be performed and build up more high-level programs such as C, Java and Python. With that said, exactly how the GPP works at a component level are confidential, restricted by the companies and what is publicly available are usually more general white papers describing functionalities.

Historically there have been two different ISA design philosophies dominating, Complex Instruction Set Computing (CISC) and Reduced Instruction Set Computing (RISC). Companies such as Intel and AMD are designing CISC computers based on the x86-64 ISA and Arm different RISC ISAs, to name the largest vendors on the market [27]. Traditionally, x86-64 processors could be found on more high-end desktop and server chips, while Arm on lower-end embedded systems with stricter power constraints, such as cell phones. Nonetheless, with Apple's latest Arm-based M1 chips and a server market increasingly buying more Arm-based chips, even the difference in targeted markets is shrinking [28, 29]. Another ISA getting attention lately is the open-source RISC-V developed at Berkeley [30].

The architecture of a GPP can roughly be divided into processing units, so-called cores and memory units. The memory units take care of the data streams on-chip, off-chip, between them both, temporal storage on-chip and more and long term memory storage off-chip. To utilise the large external memory, its long latency is hidden by creating a memory hierarchy, where smaller faster memories called caches

are located on the chip, see Fig. 2.1. The memory flow between on-chip and off-chip memory is controlled by a MCU, while other Input/Output (I/O) are taken care of with dedicated I/O-circuitry.

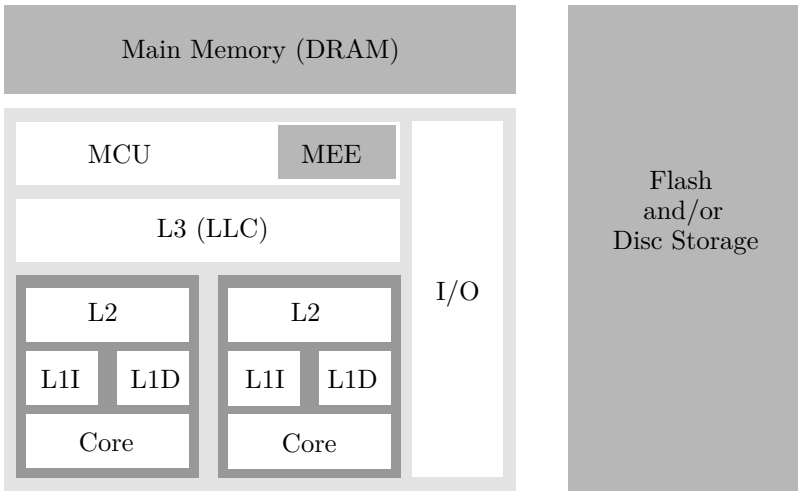


Figure 2.1: Sketch of a GPP.

Fig. 2.1 is illustrating a GPP, such as the Skylake architecture from Intel [25]. The on-chip memory is divided into three levels of cache, namely Level 1, Level 2 and Level 3 (L1, L2, L3). There exist multiple processing units, referred to as cores, possessing the execution logic. The two lower-level caches (L1, L2) are exclusive to one core and can only be utilised by it. The L1 cache can be split into two caches, one for instructions (L1I) and one for data (L1D). Lastly, the Last Level Cache (LLC), L3 in the case of Skylake, brings the cores together, enabling communication and sharing of data between them.

2.1.1 The Core

As mentioned, the cores of a GPP are where the processing of data is done. This is made possible with different functional units such as adders, shifters, comparators etc., each with specialised circuitry for its task. Today, most GPPs are multi-core and some consumer market products for server specialised applications are even getting closer to having hundreds of cores on the same chip [31]. Moreover, they are generally highly pipelined, superscalar, speculative, multi-threaded and executes instructions in an out-of-order fashion [32]. Examples of this are processors based on the Intel Skylake and Arm Cortex-M7 architectures [25, 33]. By pipelining a computer it is possible to increase the clock frequency of a core, which is done by dividing the stages an instruction has to go through, to about equal delay. A classic RISC pipeline is usually said to consist of: (1) Instruction fetch, (2) Instruction decode and fetching registers, (3) Execute, (4) Accessing memories and (5) Write back to registers, see Fig. 2.2

Today, the pipeline stages of more high-end superscalar GPP consist of more than the traditional five stages. But, all of these stages can roughly be categorised into the previously mentioned stages. Intel's Skylake architecture has between 14-19

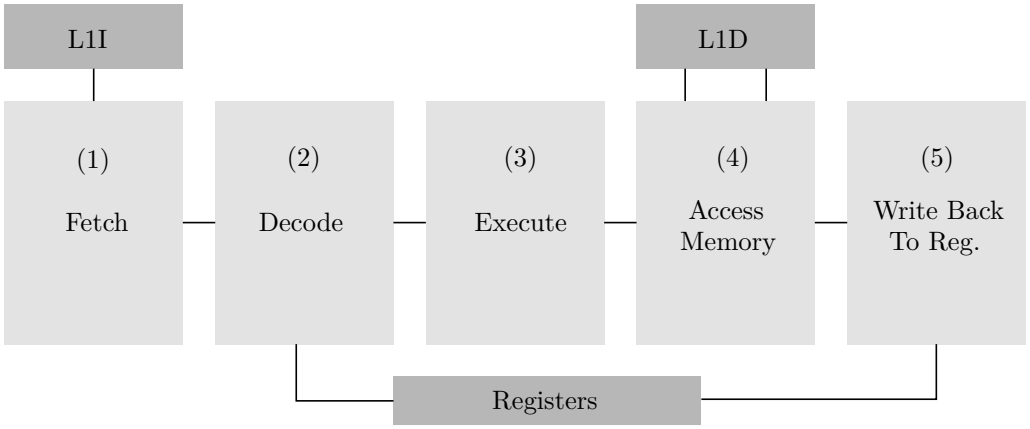


Figure 2.2: Sketch of a pipeline.

stages [25], Arm's Neoverse V1 platform 11+ stages [34], while for an example Arm's Cortex-M7 architecture has a lower number of 6 stages [33]. By being superscalar, a core can process multiple instructions at the same time, typically two, four or eight. Moreover, the core will fetch future instructions and data. By making a speculative guess on which path a branch is going, instructions in that path can be executed even before the branch is resolved. This is called that the processor is executing speculatively. If the branch is not taken, the processor will have to be flushed, stalling it. These speculatively fetched instructions will start executing instantly and stop first if the guess was incorrect. Through multi-threading a processor, multiple different processes or threads can be run on the same core, in a manner that looks to the user as parallel, effectively resulting in more programmable cores than physical. The threads can then be switched between when the running one has stalled, from a cache miss, branch misprediction, data dependency or when executed for a predetermined number of cycles. Intel and AMD call this hyperthreading and it is usually duplicate the effective number of cores a programmer can use. These threads in the hyperthreaded core, therefore, share the same resources such as lower-level caches. But registers residing in the core, are generally not shared and when a thread switch occurs, they will be offloaded. Lastly, the instructions in the core are executed in an out-of-order fashion, meaning that the core does not have to execute the program's instructions in a strict order. They will later be rearranged in the correct order in a reorder buffer, placed last in the pipeline. This was introduced in the 90s by Intel together with their Pentium pro processor and is basic practice today [35, 27].

2.1.2 The Memory Hierarchy

Ideally, a programmer would have access to unlimited sized, instantly fast memory (1 clock cycle). However, such a system is not possible to design because of chip size, wire delays, cost etc. But, by implementing gradually increased sized caches (with latency penalties per size increment) and other tricks such as prefetching, it is possible to make it behave close to an ideal memory. The different memories are structured in a pyramid-like hierarchy: (1) the smallest fastest CPU registers on

top, (2) a very small, very fast L1 cache, (3) a small fast L2 cache, (4) a not as fast larger L3 cache, (5) a slow large main memory located outside the processing chip and finally, (6) a very slow disk or a bit faster flash storage, this can be seen inverted in Fig. 2.1 [36]. These stages naturally differ with the architecture and application used. A phone does not use disk storage, but a faster smaller flash storage and might have fewer cache levels. Commercial desktop computers usually have fast flash storage such as a solid-state drive combined with or without disk storage for long term storage. Typical values for the different levels in a server computer are visualised in Tab. 2.1.

Table 2.1: Typical values for different cache and memories in a computer [27].

	CPU registers	L1	L2	L3	Main Memory
Size	1000 B	64 kB	256 kB	2-4MB	4-16GB
Latency	0.3 ps	1 ns	3-10 ns	10-20 ns	50-100 ns

The configurations of cache and memory are almost uniquely made for each processor. In this thesis, we base our model on the Skylake architecture from Intel [25]. Such a GPP has a split L1 of 32 KiB each, a 1 MiB L2 and 1.375 MiB/Core L3 cache. L1 and L2 are private to each core while L3 is shared among cores placed in the same socket, with up to 20 cores per socket.

To further hide memory latencies, multiple techniques such as reading/writing multiple data at the same time, prefetching data before it is needed, write and miss status handling register can be applied [36]. For this thesis we look more into the placement of prefetching, something that is included in most high-end commercial processors today [37]. Prefetching is a way to read data into a cache before it is needed and can be divided into two categories: software and hardware prefetching. Hardware prefetching may require extra storage usually in form of tables, keeping track of memory access patterns. Software prefetching use special instructions placed in the source code and can be set up by the compiler or the programmer. Prefetching can be located at different levels of the cache or memory hierarchy. One prefetcher commonly used today is the Stride prefetcher, which is based on the stream prefetcher. It uses first in first out stream buffers, where each stream buffer keeps track of sequentially prefetched cache lines and can so forth make qualitative guesses on what is coming next.

2.2 Trusted Execution Environments (TEE)

Numerous applications require that the GPP can ensure secure execution, separate and protected from other processes executed on the same GPP. An example of when this could be needed is cloud computing, where a client does not want other programs with malicious intention to monitor, copy or alternate the execution of their program. The most secure solution today is provided by using the so-called TEE. There are

multiple variants of TEEs available, some being Intel Management Engine, AMD Platform Security Processor, System Management Mode, Arm TrustZone, Intel SGX and AMD SEV [1, 38].

Another way to create isolation is to use VMs which can run everything from a separate process to a fully operational Operating System (OS) [1]. To manage these different VMs that share the same firmware and hardware, a hypervisor is typically used. The hypervisor has control of the software, firmware and hardware and distributes which VM can use at a certain point of time. However, an obvious flaw with such a system from a security standpoint, is that the hypervisor needs to be fully trusted. Since if it would be mounted with malicious intentions, it could exploit its privileged level to spy or mount attacks on the different VMs operating under it [11]. Another concern is that isolation is restricted to the processor implementing them. Other units controlling the bus such as Direct Memory Access (DMA) controllers or a Graphical Processing Unit (GPU) could potentially bypass this isolation. According to [11] so is virtualisation also ignoring security threats coming from hardware attacks, targeting debug and test the infrastructure of the chip.

By using hardware-assisted TEEs, the intention is not to only offer greater security but also higher-performing systems. The TEE will generally keep an area of the GPP restricted to some process, where this process can access other processes area but not the reversed. We provide in the following sections a deeper explanation of how TEEs from three of the largest GPP designers function, namely SGX from Intel, AMD SEV and Arm TrustZone.

2.2.1 Intel Software Guard eXtension (SGX)

Intel's SGX is one of the most extensively used TEEs today. It was first introduced for client platforms with the Xeon E3 processor and got extended for server use [39]. With SGX, Intel is trying to enable secure remote computing [9]. It works by creating exclusive memory regions called enclaves, a region that the CPU will protect from all accesses coming from outside this region [9]. This enclave has therefore a restricted memory region for its use, which is a subset of the main memory. On-chip it holds the enclave's data and code in 4 KiB pages also restricted to its use only. By using a software attestation scheme, a remote party can authenticate that the software is running as intended, inside the enclave. Since the SGX is only trusting the GPP's internal components, a central part of the SGX is the use of an MEE to keep the enclave encrypted with 128-bit key Advanced Encryption Standard (AES) in the off-chip memory [12]. A deeper explanation of the MEE is outlined in Section 2.3. Not only does SGX require the enclave to be encrypted in GPP's external components, but the main memory also needs to be both replay protected and tamper-resistant. By keeping the memory space of the enclave exclusive it should be protected from internal attacks. However, as Costan et al. mention in [9], the MEE does not protect addresses of the cached enclave pages in the main memory. Therefore with carefully crafted malicious software, pressure could be put on the LLC and tap its content from observed memory accesses. There have also been multiple attacks targeting the SGX that will be discussed later in Section 2.4.

2.2.2 AMD Secure Encrypted Virtualization (SEV)

The SEV provided by AMD is also designed for secure cloud computing [10]. It works by encrypting VMs in main memory and according to Kaplan et al. [10] so can this not only protect from physical threats but also other VM's working on the same hardware, even the hypervisor. The main memory encryption is done with dedicated hardware in the MCU, using a 128-bit AES key randomly created at boot time, the same as SGX. When encryption is enabled, an extra bit will be added to the physical address called the C-bit keeping track if a page is encrypted or not. The SEV can be configured to occupy VM machines spanning over one or multiple cores.

Traditionally a hypervisor and its VM works as a ring-based security model. The hypervisor is more privileged and can therefore access all resources in the VM but not the other way around. But in SEV the different levels will be isolated and the hypervisor can only communicate with the VM in controlled, restricted communication paths. As mentioned, security outside the chip is done with encryption, on-chip it is done by using tags for isolation between VMs. The VM can choose to only encrypt a subset of pages which can make another subset open used for communications with hypervisor and other VMs.

To provide confidentiality for the owner of the guest VM, such as a customer in a cloud computing system, the guest owner will first provide a guest image to the SEV firmware. A launch of the guest is then done and measurements of this are sent back to the guest owner. If the guest owner deemed the measurement correct then it can provide extra resources needed for computation. The measurements are an authentication of the platform that it has an AMD SEV done with a specific identity key and an attestation that the guest was launched with SEV enabled.

2.2.3 Arm TrustZone

Arm's version of TEE differs a bit from Intel's and AMD's. Rather than providing one complete solution, they are providing a range of different components that the chip designer then can choose from [11]. Security is provided by dividing up execution into two worlds, one named secure and the other normal. Components used in the secure world are not accessible by the normal world. There are architectural extensions that allow the same core to execute code from both worlds, switching between the two in a time-sliced fashion. In an Arm-based GPP where TrustZone is enabled, it will always start in the secure world before running any normal world sequences. This is to guard the system against system boot attacks and such it can authenticate that the secure world is secure.

TrustZone works by partitioning software and hardware resources to be in either a secure or normal world [11]. Memory units like caches and main memories can consist of data from both worlds, but with extra tags, the controllers such as the L2 cache controller and DMA controller have separate channels for data belonging to the secure or normal world. Compared to SGX and SEV so is the data not encrypted in the main memory, but only strictly isolated. Processor cores each consist of two

virtual cores, one for secure execution and one for non-secure. To keep track of when a unit is operating in the secure or normal world an added bit is put to the buss and cache memories tag's address. The so-called non-secure bit is added as a control signal to the read and writes busses such that the different units can keep track if a master's access to a slave is eligible, if not, an error on the bus will be raised.

Important to note, as a comparison between TrustZone, SGX and SEV, is that Arm is an Intellectual Property (IP) provider, offering an extensive set of IP blocks capable of using TrustZone. Different systems use different setups of these IP blocks. When labelling a system with Arm TrustZone no consideration is taken into which units or how many actually can make use of TrustZone and therefore cannot be assumed to act similarly under the same threat model [9].

2.3 Memory Encryption Engine (MEE)

As described in the previous section, some TEEs are leaving the main memory untrusted. They, therefore, need to make use of some encryption mechanism to keep data and instructions in the main memory encrypted from potential threats. To not lose performance and keep the security high this is usually made possible with autonomous hardware-accelerated MEE [9, 12, 10]. The MEE is located inside the MCU and by keeping track of when the MCU refers to a protected area in the main memory (such as SGX enclave, or C-bit in AMD's physical address) read/write requests can be routed through the MEE to encrypt/decrypt the data accordingly. In [12] Gueron describes the technologies that the MEE design of SGX for Intel's Skylake architecture was based on. He there mentions that it "is based on the following pillars: an integrity tree, the cryptographic primitives that realise the encryption, the Message Authentication Code (MAC), and the anti-replay mechanism" [12, p. 2].

The integrity tree is used for fast verification of the data's integrity and is a common method for keeping track of a large amount of data with limited storage. The type used is the classical Merkle Tree, a tree-like hash structure where each node keeps a hash of its two children and the lowest nodes keep the protected data. However, instead of hashes, MAC tags are used. The root of the tree will be stored internally of the chip while the rest in the main memory. The integrity tree can henceforth be used to ensure the integrity of a memory block and that it has not been tampered with since the last access. This checkup of the integrity tree will add on latency for encryption/decryption proportional to the size of the tree.

When using an MEE, an encryption standard needs to be implemented. In the aforementioned MEE, AES with a key length of 128-bits is used [12]. The AES is a standard established by the U.S National Institute of Standards and Technology in 2001. It requires 10, 12 or 14 rounds for encryption with 128, 192 or 256 bit key length respectively [40]. In the Skylake architecture, in the fastest scenario, the MEE needs to do 15 cycles for one write, 5 AES operations and 10 polynomial multiplications [12]. This is done with dedicated hardware in the MEE, separated from the core's logic. This MEE has a throughput of 1 AES block per cycle and a 32 GiB/s limited to a maximum clock speed of 32 GHz.

2.4 Attacks

While the TEEs promise secure and trusted execution environments there have been numerous publications exploiting them. Since most published attacks are towards Intel's SGX, we focus on some of these to understand their concept. Even if they focus on SGX they bring up interesting weaknesses that could affect AMD and Arm TEE based systems as well. In [13] Nilsson et al. divide the known published SGX attacks into the following categories: controlled channel attacks, cache-attacks, branch prediction attacks, speculative execution attacks, rogue data cache loads, microarchitectural data sampling and software-based fault injection attacks.

Most of these attacks fall under the category of some type of side-channel attack, which is a way of exploiting observable side effects that is a result of computing. There have been multiple attacks where physical side effects such as acoustic noise [41], power consumption [42] or electromagnetic radiation [43] have been observed to extract secrets such as cryptography keys. Note that these are not attacks targeting TEE, but logic chips in general. These types of attacks have also been proven to work on more sophisticated hardware's such as personal computers [44]. Since GPP often compute code from unknown origin, such as cloud computing or as a general PC user, downloading software from the internet, these systems therefore also face threats of software-based attacks.

Another use of side channels is the cache-based timing attacks where the cache-hierarchy system is used to observe memory access patterns. Usually from a malicious process running in parallel with the secure environment, such as the same core in hyperthreading or a parallel core. There exist many different types of these such as the Prime+Probe [45] and Flush+Reload [46] attacks. SGX has been proven to be vulnerable to similar cache-timing attacks based on the same concepts, one example being the CacheZoom attack based on Prime+Probe [16, 18, 19].

To get an understanding of how the cache-based timing attack works we describe the Flush+Reload technique [46]. As an optimising technique, different processes can share the same pages if those consist of the same data to save physical memory space. If we then create a spy process that has an equal page as the victim, then the GPP could keep these two virtually different pages as the same physical one in the cache, to save memory space. If the spy then tracks accesses of its page, sensitive data from the victim could leak. More specifically, this is done by flushing a specific cache-line, effectively removing the content from the cache (it will still be in a higher level memory such as a higher cache or off-chip memory). Then let the victim operate for a while, to finally as the spy trying to access the same cache-line. By measuring the time this access takes it is possible to know if it was a cache hit or miss. If hit, then we can conclude that the victim process has used this cache line specifically because it would have to bring the cache-line from a higher level memory. To get the spy to share the page with the victim Yarom et al. [46] located first the victim's executable files and then mapped them into the virtual space of the spy, tricking the GPP into the sharing page optimising effect. This assumes though, that the spy has access to the same program code as the victim.

By making use of the control that an untrusted OS can have over a platform (such as an OS in cloud computing) other side-channel attacks could be launched. One such attack is the one performed by Xu et al. [47], by observing page faults patterns of a program running outside a TEE, they could then extract information when the same program ran inside a TEE. They showed it possible, among others, to extract parts of JPEG pictures in a JPEG compression program. Another such attack exploiting the OS privilege level is the SGX-Step attack, single-stepping through an enclave's instructions [48]. It does this by configuring timer interrupts using Intel's advanced programmable interrupt controller and track table entries from user space.

In 2018 both the Spectre [21] and Meltdown [22] attacks that both exploit the speculative, out-of-order execution of a modern GPP made headlines. More specific, the processor speculatively execute instructions before a branch earlier in the instruction flow has been determined. Both of these attacks makes it possible to trick the GPP to prefetch data and execute some instructions that should not have been done in a linear execution flow. Following this, different techniques can be used to extract sensitive information, one of which using cache side-channel attacks such as the Flush+Reload previously described. Moreover, both of these attacks have been built upon by other attacks targeting SGX, with Spectre-like attacks such as the SGXPectre [49] and the SpectreRSB [50].

The most famous Meltdown lookalike targeting SGX might be the Foreshadow attack [20]. Different from the previously described attacks, Foreshadow does neither require any knowledge of the enclaves source code nor to exploit any vulnerabilities from software. Moreover, it does not require any cache-based side-channel, as exploited in the other attacks. The Meltdown attack shows susceptibility in Intel's speculative out-of-order GPP. When a process makes access to a memory location it has no read right to, a fault will be issued and taken care of by an exception handler. But while this fault is being taken care of there will be a gap of time where consequent instructions can be issued. By loading an oracle memory location (predetermined by the attacker) based on the value from the restricted data just read, the secret data can then be obtained by measuring the time it takes to reload the memory slot of the oracle. By exploiting this fundamental of the Meltdown attack and adding tricks to step through the secret data that resides inside the enclave's L1 cache. Then the full data memory inside a secure enclave can be extracted.

This thesis proposes a deeper integration of the MEE which has the potential of making these types of attacks obsolete. More of this will be discussed later in the Section 6.1.

3

Proposed Systems

In this thesis, we take a broader look into integrating the MEE deeper into the processor's memory hierarchy and how this affects its performance. We see two systems that could potentially increase security in the system compared to today's TEEs (both visible in Fig. 3.1): Both of these systems prove to have small performance losses

- In the first system, the MEE is placed before the LLC. Here it could be chosen to only encrypt data, instructions or both.

- In the second system, the MEE is placed before the L1D cache. To hide the overhead of the MEE this system is also suggested to be fitted with a smaller dedicated encryption cache devoted to the secure environment, we refer to it as L0D. This cache can reduce perceived delays for the core and could be fitted with prefetching techniques to start decryption before data is needed in the core.

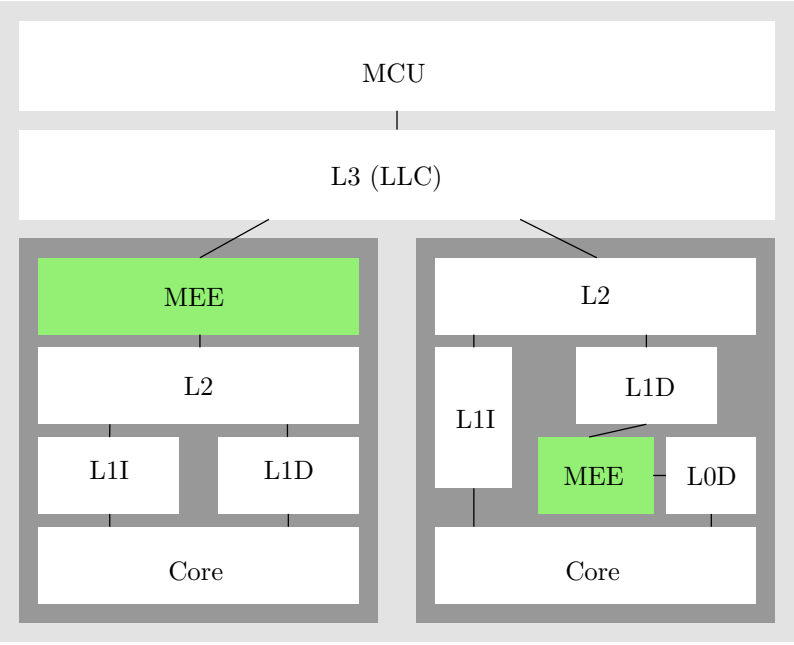


Figure 3.1: The two proposed systems.

These systems will offer different levels of security depending on additional design features and how security is defined. With no multithreading in the cores, encrypted data in the LLC could be sufficient. But if the core has multithreading, then either all threads running on the same core would have to be assumed to be non-malicious to each other. To not having to do this conjecture, the addition of an L0D cryptographic cache and keeping it restricted only to the secure threads could do it.

4

Experimental Methodology

This chapter goes through the experimental base system that was configured in the gem5 simulator to simulate the proposed systems. As binary for the gem5, the SPEC cpu2006 benchmarks were chosen and a brief discussion about these is also provided.

4.1 Simulated system

A base system representing a modern single core computer, inspired by Intel's Skylake architecture, was defined according to Tab. 4.1, representing the system in Fig. 2.1 [25, 51]. Other configured and tested systems were all based on this system but deviated from it by architectural reorganisations, extensions, and/or alternated parameters.

CPU	1 core, x86-64, 3.2 GHz, OOO, 1 ROB with 224 entries, 8 fetch/decode/rename/IEW width, 14 stage pipeline
L1	64 KiB, 8-way set-associative split I/D, data=4, tag=2, response=4 cycles latency 10 MSHR x2, 8 write buffers x2
L2	1 MiB, 16-way set-associative, exclusive, data=14, tag=4, response=14 cycles latency stride-prefetching 20 MSHR, 16 write buffers
L3	2 MiB, 16-way set-associative, inclusive, data=50, tag=8, response=50 cycles latency 40 MSHR, 32 write buffers
MCU/ Main memory	55 ns, 12.8 GB/s bandwidth, including delay both for MCU and DRAM
MEE	Encypt/Decrypt 15 cycles latency, no limiting bandwidth

Table 4.1: System simulated in gem5.

4.2 The gem5 Simulator

The open-source gem5 simulator was chosen for this thesis [24]. It is a merge between the M5 and GEMS simulators, providing simulation with the most common ISAs and a good range of different CPU models. The simulator can operate different types of CPUs in either System-call Emulation (SE) mode or Full-System (FS) mode. The SE mode is avoiding the need to emulate a full OS compared to the FS mode that operates this by separating user- and kernel-level instructions and modelling all of the OS's devices. As only one executable file (one benchmark) at the time will be executed it was chosen to run gem5 in SE mode.

Moreover, the gem5 simulator offers two different memory system models. First being named Classic, which gives an easily configurable, fast modelled system and was the one chosen for this thesis. The second one is named Ruby and can simulate more accurate caches. It is of great use for those wanting to simulate things such as cache coherence protocols.

Gem5 provides multiple different CPU models and the O3CPU (Out-Of-Order) was chosen. The model will simulate "dependencies between instructions, functional units, memory accesses, and pipeline stage" [24, p. 5]. The pipeline consists of the stages: Fetch, Decode, Rename, Issue/Execute/Writeback and Commit. By setting how many cycles it will take for an instruction to move from one stage to the next we can specify the length of our pipeline. The O3CPU model does what Binkert et al. call "execute-in-execute", meaning that instructions will only really be executed first when dependencies have been sorted out [24].

Exact parameter values set for the base system, including those given in Tab. 4.1, can be found in Appendix A.3.

4.3 Benchmark Applications

To compare the different systems the SPEC cpu2006 benchmark suite was chosen [26]. This suite consists of 30 benchmarks, which can be divided into integer or floating point. The benchmarks used were:

400.pelbench	*401.bzip2*	*403.gcc*	*416.gamess*
429.mcf	*433.milc*	*435.gromacs*	*436.cactusADM*
444.namd	*445.gobmk*	*450.soplex*	*437.leslie3d*
453.povray	*454.caclculix*	*456.hmmer*	*458.sjeng*
459.GemsFDTD	*462.libquantum*	*464.h264ref*	*465.tonot*
470.lbm	*471.omnetpp*	*473.astar*	*482.sphinx3*
998.specrand	*999.specrand*		

The benchmarks: *447.deaII*, *481.wrf*, *483.xalancbmk*, could not be used due to failure in compilation or error during the run. How these benchmarks were compiled and how they were chosen is described in Appendix A.2. They were all run with the

provided reference input and arguments for each benchmark were set according to [52].

All benchmarks were used for the first run simulating the most basic systems. By extracting data of cache misses from the base system Misses Per Kilo Instructions (MPKI) could be calculated which can be used as a measurement to rule out benchmarks that have more cache misses. The benchmarks chosen from this criteria were: *401.bzip2*, *410.bwaves*, *436.cactusADM*, *437.leslie3d*, *445.gobmk*, *458.sjeng* and *470.lbm*. More of this are discussed in Section 5.1.1.

Each application was simulated for 10 million instructions with a 10 million instruction warm up. Warming up the caches by executing a part of the program before taking measurements gives values closer to the expected average of the program [53]. If measurements had been started from the beginning of a program, a larger portion of the measured time would be caused by stalled processor due to cache misses. To achieve the 10 million instruction warm-up two simulations ran for each test. One with 20 million and a second with 10 million instructions. The simulated time for 10 million instructions with warmed up caches was then calculated as the delta between the two. The simulation time for each benchmark is presented normalised to the base system without any encryption, i.e. a normal GPP without any encryption stage.

In this thesis we simulated the six following systems:

1. Base system with no encryption used to normalised data to.
2. MEE placed before off-chip main memory encrypting data and instructions, representing Today's TEEs.
3. MEE placed before LLC encrypting both data and instructions.
4. MEE placed before L2 encrypting only data.
5. MEE placed before L1D.
6. MEE placed before L1D but with an added L0D. This was investigated further with prefetching.

System 1-5 is referred to as naive systems since they naively place the MEE at different places in the memory hierarchy. System 3 is the first proposed system and system 3 the second proposed system, discussed in Section 3.

5

Results

This chapter provides results from the simulations and a discussion around those.

5.1 Naive Systems

As today's TEEs have their encryption stage, their MEE located inside the MCU, between the LLC and main memory, an interesting analysis to be made is how performance is affected by moving this encryption stage to other locations in the memory hierarchy. Since these systems are simply a matter of moving the MEE without any more consideration, we refer to them as "naive systems". The results for placing the MEE before the main memory, LLC, L2 (only encrypting data), and L1D can be seen in Fig. 5.1.

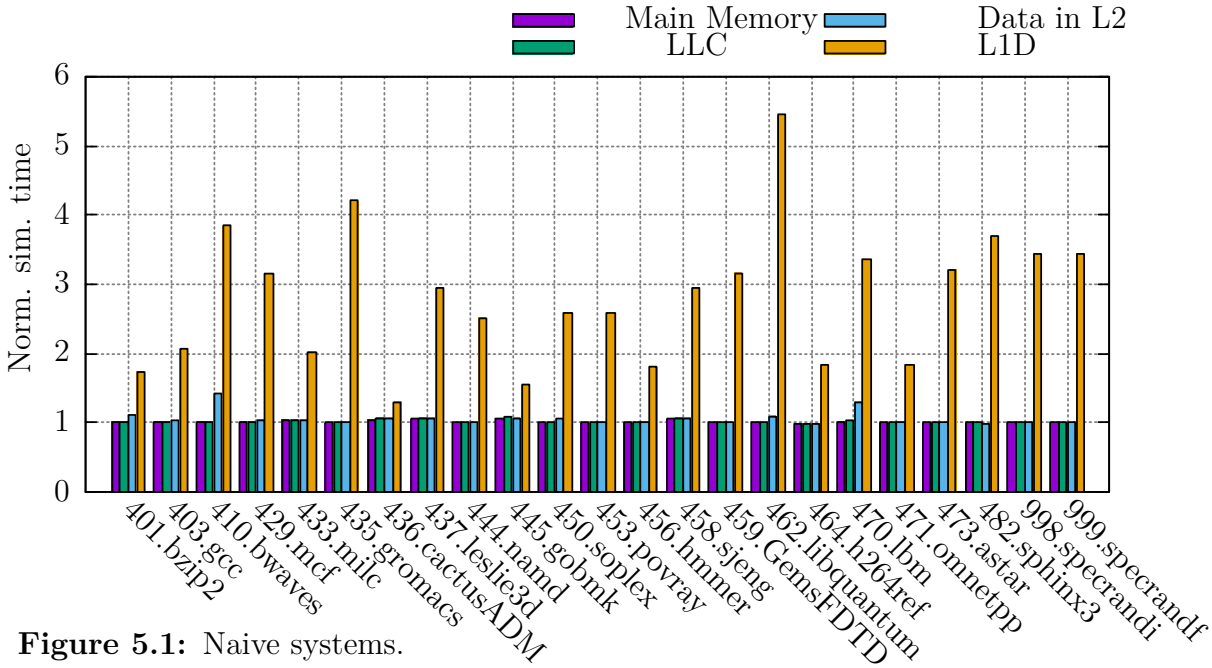


Figure 5.1: Naive systems.

The graph is illustrating four systems (system 1, 2, 3 and 4 introduced in Section 4.3), all based on the base system in Tab. 4.1. The first, most left one system (purple bar) corresponds to the modern TEEs such as SGX and SEV that have their MEE located before the main memory. The second system (green bar), is with

the MEE stage moved to before the LLC instead. The third system (blue bar), is with the MEE placed between the L1D and L2, resulting in only keeping the data (but no instructions) of the system encrypted. The last system (orange bar), is if the MEE is located directly between the core and L1D, encrypting again only data. All of the systems are normalised to the base system without any encryption giving a y-axis of how many times slower it was compared to the base system for this specific benchmark.

As one can suspect, naively adding an overhead, effectively increasing the time it takes to read and write decreases the performance of the system. Interestingly though is that the effect is not that great for the L2 and LLC systems, while it is significantly more for most of the benchmarks with an encryption stage before the L1D cache. This suggests that from a performance perspective the MEE could be naively more deeply integrated until before the L2 cache. However, simply starting encrypting data in the L1D cache could have a serious impact on the performance of the system. Therefore further additions to such as system would have to be made.

5.1.1 Running Less Tests

Because the simulated MEE just adds an overhead between two memory stages, the most interesting benchmarks are those with more memory accesses. With more memory accesses, the total execution time of a constant set of instructions should be more sensitive to an added overhead between two memory stages. This, since if there is no data or instruction available at the GPP's execution unit because it is taking time reading it from memory, then it will have to stall, consequently reducing the total performance of the GPP. If the memory write action takes longer then the processor could have to discard a memory read access since the data written on that specific location is not up to date. This could take effect in that either the core would have to do another read request or the first read request would have to wait until the data is up to date and ready to read, effectively increasing total execution time.

To distinguish these benchmarks, the Misses Per Kilo Instructions (MPKI) can be used for comparison. Results for such a test of memory accesses in LLC can be seen in Fig. 5.2.

In this plot, only data from the base system with no encryption were considered. MPKI are represented on the y-axis and benchmark name on the x-axis. A higher MPKI is to be interpreted as a system with longer processor stalls caused by misses in the LLC. Therefore will the total simulation time be more sensitive to the added read time to main memory, making it more sensitive if this would increase. In the case of this thesis, the benchmark with more memory accessing activity should be the ones affected by an added latency from an encryption stage. Therefore these benchmarks were distinguished and for the rest of the simulations, only the benchmarks: *401.bzip2*, *410.bwaves*, *436.cactusADM*, *437.leslie3d*, *458.sjeng* and *470.lbm* were used. This decision was based on the results from Fig. 5.2 and the criteria to have an MPKI above 10. The choice of an MPKI of 10 was based on the

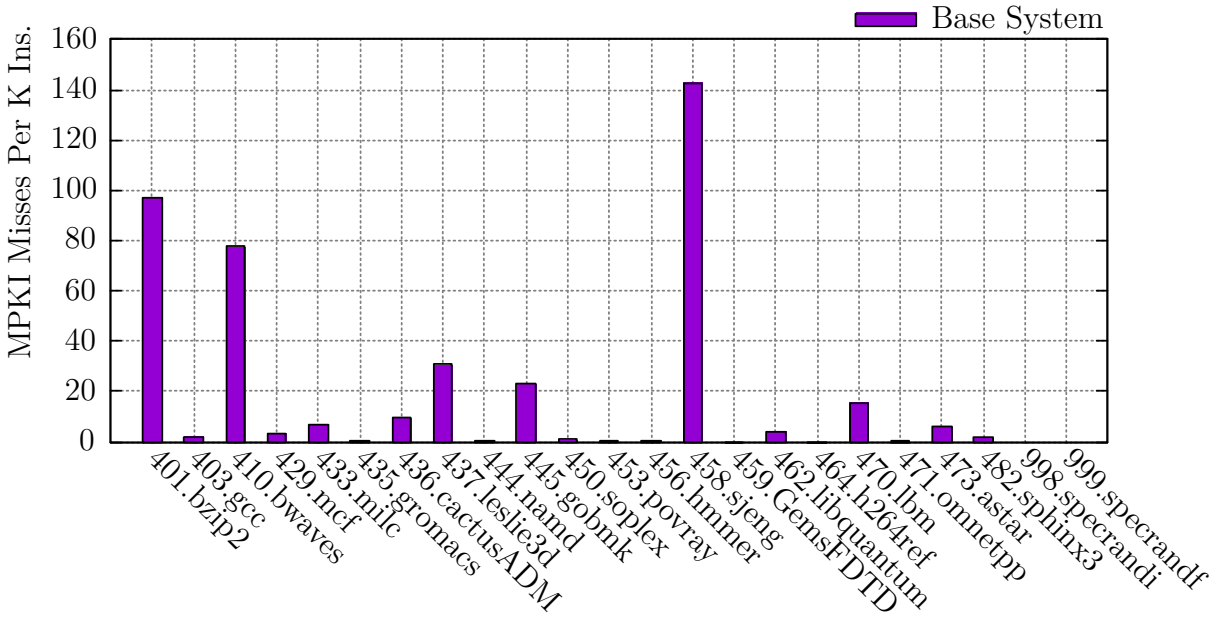


Figure 5.2: MPKI for LLC, base system.

following arguing: a miss in the LLC leads to a total memory read time in order of 10^2 processor cycles (sum up L1, L2, L3 and main memory access times in Tab. 4.1. An MPKI of 10 indicates therefore that for 1000 cycles where the processor could operate without stalling, it would have to stall for about another 1000 cycles, waiting for a read in main memory corresponding to an execution time where half was spent on waiting for data and half executing it.

5.1.2 Placing the MEE Before the LLC

In a three-level cache system where the two upper-level caches (L2 and L1) are restricted to one core and if this core is running either a single thread or all threads could be assumed to exist inside the same secure environment, then an interesting case would be to move down the encryption stage to before the LLC. Hence keeping data and instructions encrypted inside the shared LLC and therefore restricting plain text knowledge of the secure thread or threads executing on that core. In Fig. 5.3 data are presented with simulations using different encryption and decryption delays for such a system. It shows five different systems, all with the MEE placed between L2 and LLC. From left to right the encryption/decryption delay is set to 10, 15, 20, 25 and 30 cycles. A system of 10 cycles could represent a system with 128 bit AES that one of each 10 rounds needed for the transforming is done each cycle and no other specific hardware is needed that adds extra cycles. As outlined in Section 2.3 so will the MEE in Intel's Skylake in best case take 15 cycles for 128-bit AES, which is represented in the green bar. Moreover, this was an ideal case and when the integrity tree takes up more time to go through it the cycle count will be higher, increasing the number of total cycles. Also, the more complex 192-bit and 256-bit AES keys take more rounds to transform, 12 respectively 14, adding more cycles if they would be implemented instead. Therefore, simulations for MEE with 20, 25 and 30 cycles MEE were also simulated. All the systems are in Fig. 5.3

normalised to the base system with no encryption.

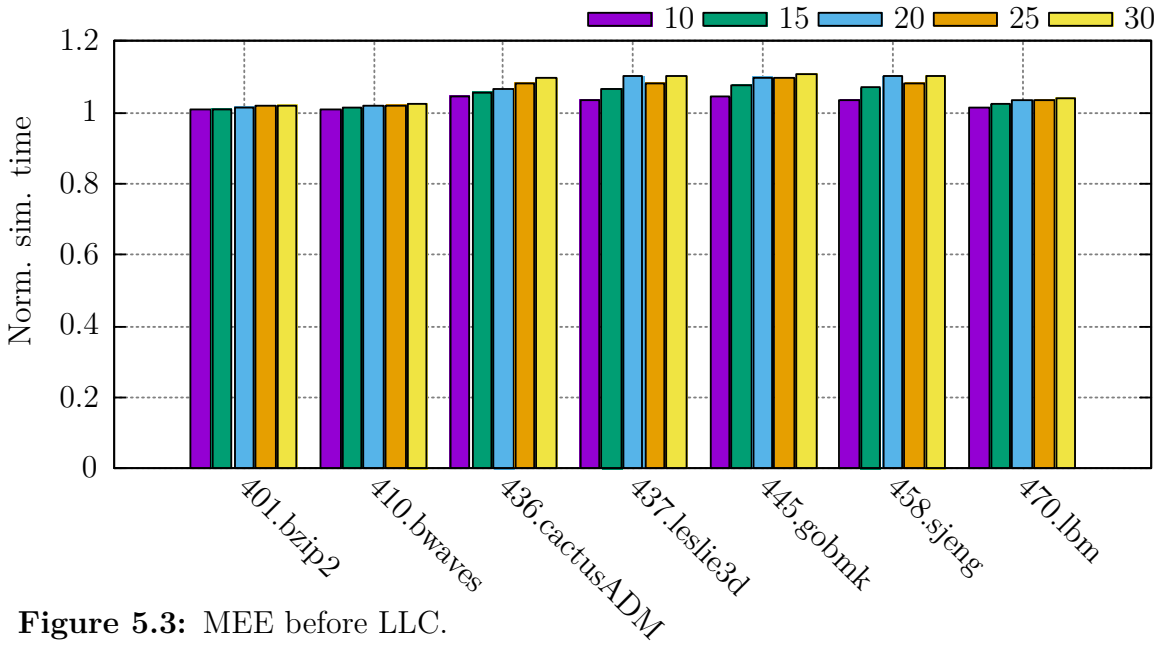


Figure 5.3: MEE before LLC.

From Fig. 5.3 an expected increase performance toll can be observed from increasing the encryption/decryption latencies. What can be noted is that it is low, not going over a 10 % of the non-encrypted base system for a 30 cycles delay. As can be seen in the same Fig. 5.3, benchmarks *437.leslie3d* and *458.sjeng* produces unexpected results for simulations with 25 cycles encryption/decryption latency. This behaviour is hard to explain, but a qualitative guess is that it could depend on some timing anomalies were, for a dynamically executed processor, cache misses do not necessarily result in a slower system [54].

5.1.3 Placing the MEE Before the L1D

Another interesting system to look closer into is one with an encryption stage before the L1D cache. If such a system restricts the encryption key used to one thread running on the core, then all other threads running on that same core would not be able to interpret its data in L1D. Results from simulations running with different encryption and decryption latencies can be seen in Fig. 5.4. Same as the previous Fig. 5.3, Fig. 5.4 shows five different systems, 10, 15, 25 and 30 cycles encryption/decryption, all with the MME stage moved to between the core and L1D cache. The data are normalised to the base system with no encryption.

Just as in Fig. 5.3, the results in Fig. 5.4 are showing a decrease in performance of systems with added latency to memory reads and writes. However, adding the same overhead to the memory stage before L1D has a significant toll on the system performance compared to placing it before the LLC.

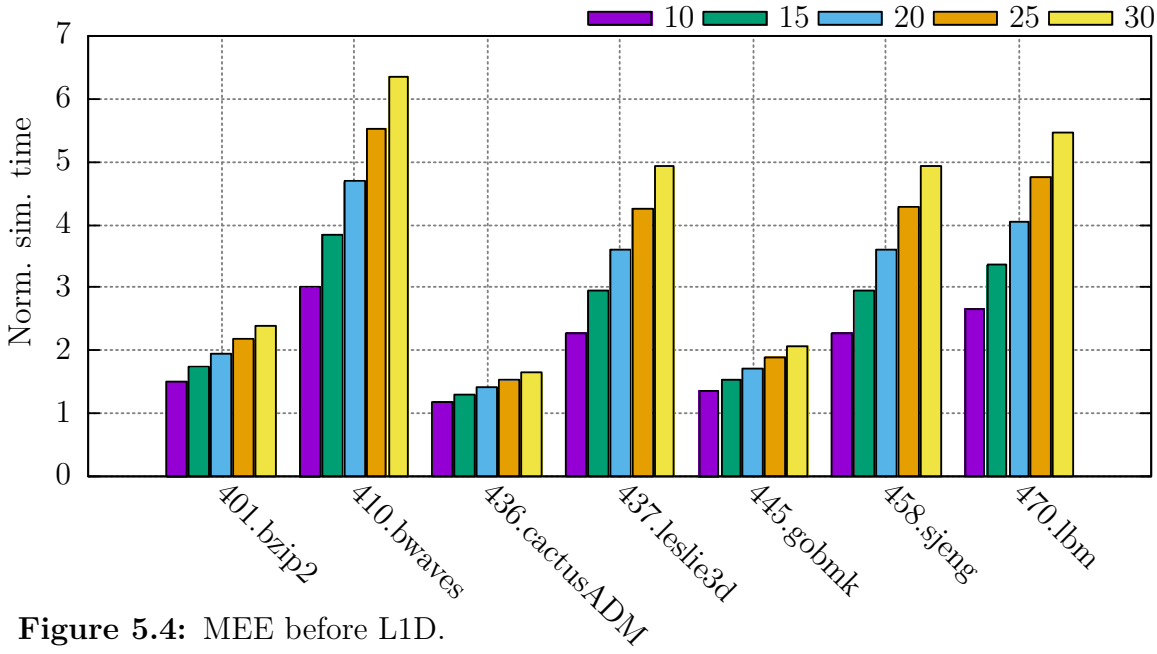


Figure 5.4: MEE before L1D.

5.2 Hiding Encryption Overhead With a Dedicated L0D Cache

As seen in Fig. 5.4, adding an encryption stage before the L1D have a significant negative impact on the system's performance. To decrease or hide this performance loss, one possibility is to add a smaller cache before the encryption stage (when placed before the L1D cache) and dedicate it to only the secure thread or threads. This is the already discussed L0D cache whose placement can be seen in Fig. 3.1.

First, we start by analysing how the size of an L0D could impact the system's performance. In Fig. 5.5 results from simulations with five different sizes of L0D can be observed, all with an encryption/decryption delay of 15 cycles. The sizes chosen for simulation were 16, 8, 4, 2 and 1 KiB. An L0D sized 16 KiB is half the size of the 32 KiB L1D and it was judged that a larger cache would probably not be an attractive option. In Fig. 5.5, the left-most system, with a purple bar is the naive system with no L0D and the MEE placed between core and L1D, added for reference. The same system can be viewed in previous Fig. 5.1 and Fig. 5.4.

From the results, it is possible to see that for most benchmark, even a small L0D cache of 1 KiB could be sufficient to reduce the performance loss caused by an encryption stage. Only the *410.bwaves* benchmark shows a clear dependency on the size of the L0D cache. As a 1 KiB cache was judged to be a considerable small cache already, no smaller cache sizes were investigated.

5.2.1 Increasing Performance With Prefetching

In all systems until this point, a Stride prefetcher has been located at the L2 cache, for the full system see Tab. 4.1. A compelling question is if the overhead of encryp-

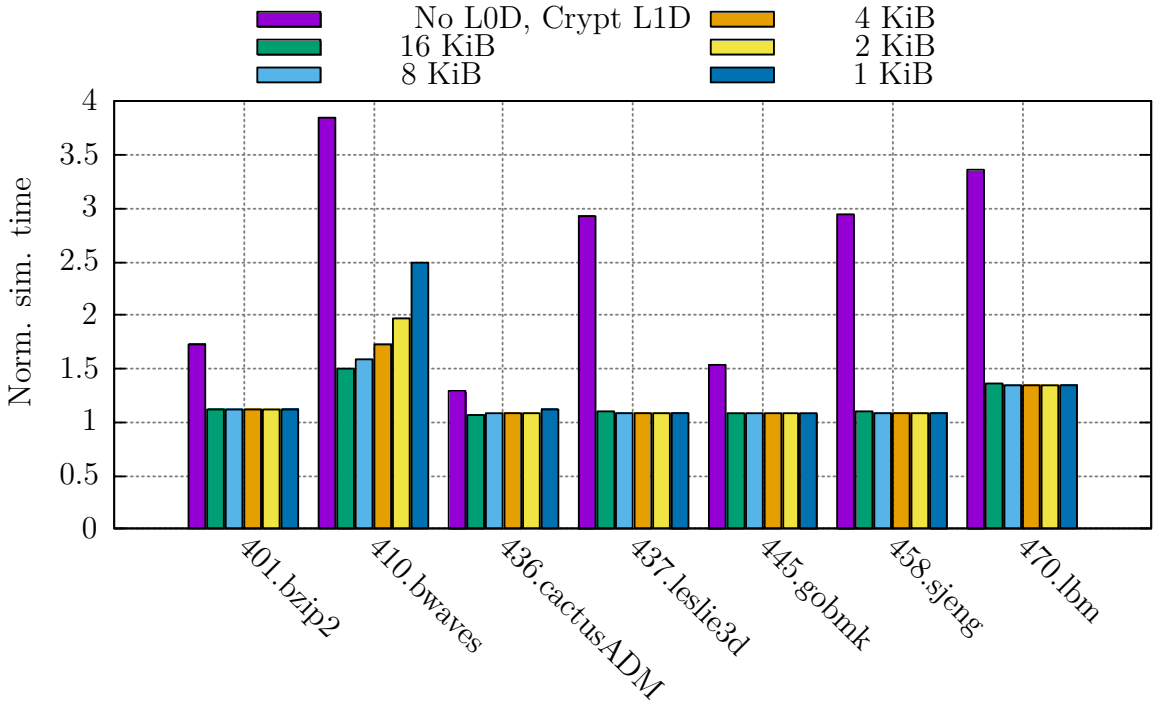


Figure 5.5: Adding a dedicated L0D cache for sensitive threads.

tion could be reduced even further from what an added L0D cache already gives. We therefore experimented by moving the existing stride prefetcher from L2 to L0D and another system with both a stride prefetcher at L2 and L0D. Results from those systems can be seen in Fig. 5.6 where a 1 KiB L0D was used. These systems are the orange and yellow bars respectively. In the same figure, one system with no prefetching at all (blue bar), one with no prefetching and an encryption/decryption delay set to zero (green bar) and lastly the naive system with no L0D and the MEE placed between core and L1D (purple bar) were all added. All systems have an encryption/decryption delay of 15 cycles, except the green bar that is set to zero, as mentioned earlier. Moreover, all results are normalised to the base system with no encryption nor L0D cache. In Fig. 5.6 the benchmark *410.bwaves* are absent since the simulation could not finish due to errors in the simulator.

As can be observed from the graph, disabling prefetching completely will decrease the performance of the system (blue bar). No difference between having one prefetcher only (orange bar), located in L0D can be seen compared to the previously mentioned system for benchmarks *436.cactusADM* and *458.sjeng*. However, for most benchmarks, a significant improvement can be seen where even for some benchmarks, namely *445.gobmk* and *470.lbm* the system has a better performance than for the base system with no encryption, no L0D and a stride prefetcher at L2. This can be seen through the value of those systems being less than 1. By adding another prefetcher at L2 (yellow bar), which also the base system have, all benchmarks except *436.cactusADM* show an even further performance gain.

Moreover, by comparing the system with encryption/decryption delay set to zero (green bar), with the same system but a 15 cycles delay (blue bar) it can be observed

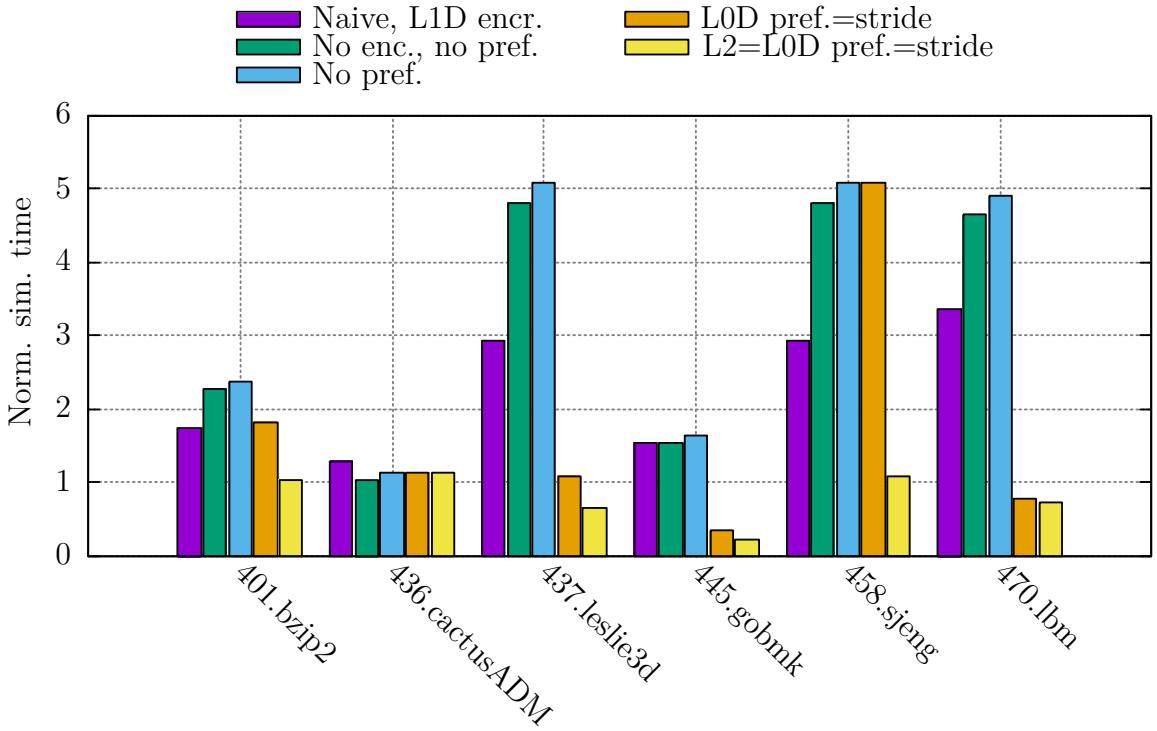


Figure 5.6: 1 KiB L0D, with prefetching at different stages.

that the two systems simulated compare to each other as expected. By adding an encryption delay the system's performance will decrease, however, not significantly since it is using an L0D cache that reduces the performance loss, which we could already see in the previous Fig. 5.5. Comparing the Fig. 5.5 with Fig. 5.6 it is also possible to detect that moving the prefetcher located in L2 to L0D does not necessarily increase performance. Benchmarks *401.bzip2* and *458.sjeng* show even a decrease in performance, rather drastically in the latter, suggesting that for those benchmarks having the prefetcher in L2 instead is preferable. Nonetheless, benchmarks *445.gobmk* and *470.lbm* challenge this by giving an increase in performance by moving the prefetcher, while the not already mentioned benchmarks *436.cactusADM* and *437.leslie3d* shows no difference. This suggests that if only one prefetcher is to be included in the system, then a consideration of what type of load the GPP would be exposed to should be weighed into the decision of where to put it. Lastly, if there is a possibility to add a prefetcher to both locations this should be the most attractive option.

Another simulation regarding prefetching was conducted. Still with the same systems as in Fig. 5.6, but with a 16 KiB L0D cache instead of the previous 1 KiB. The results can be seen in Fig. 5.7. For this run, neither *410.bwaves* nor *438.leslie3d* could complete due to errors and are therefore not included.

The results in Fig. 5.7 are more or less identical to those in Fig. 5.6. If one looks back at the results from Fig. 5.5 can it be observed that most of the benchmarks did not show any significant performance differences between the different sizes of L0D, something that could explain this similarity. The same discussion made for the results in Fig. 5.6 could therefore be done to these results.

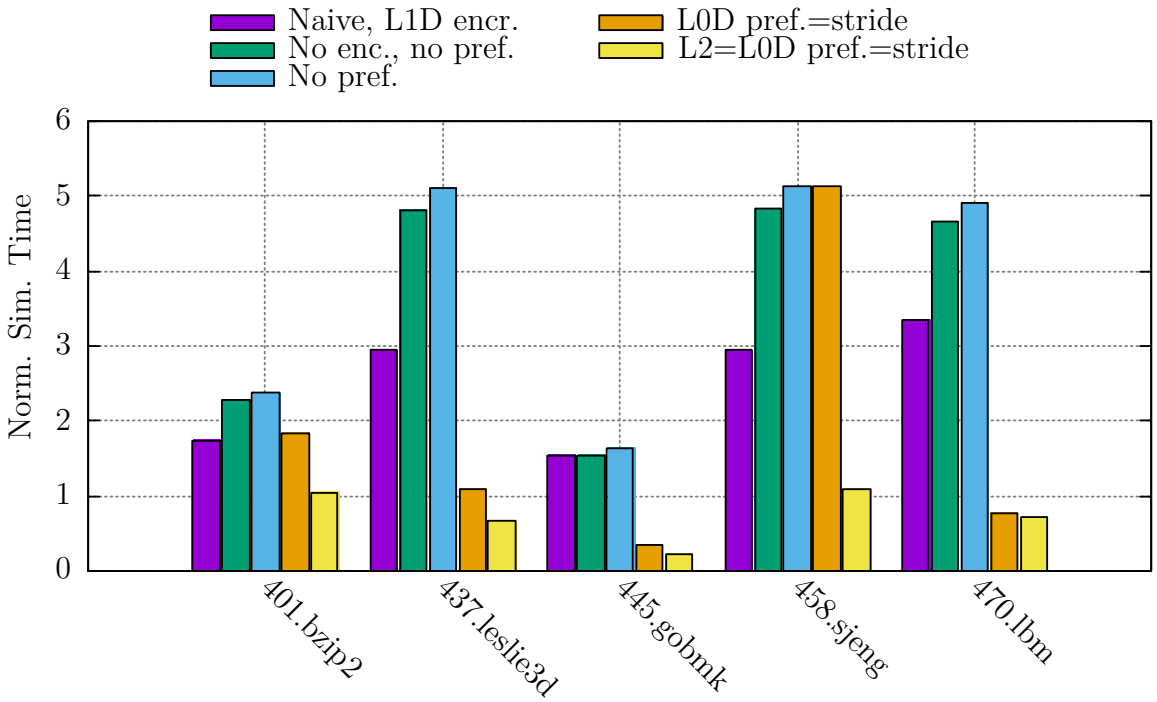


Figure 5.7: 16 KiB L0D, with prefetching at different stages.

6

Concluding Discussion

6.1 Discussion

Having access to secure systems is something that is today highly regarded which will probably increase in both the near future and beyond that. Different types of architectural secure systems, with different level of actual protection, are offered today. One could divide those into a top and bottom tier, where the TEE with their dedicated hardware for encryption and isolation are placed at the top one and VMs in the bottom tier because of their natural submissive relationship with the untrusted hypervisor. However since even the top tier, secure environments cannot offer total security (outlined in Section 2.4) this thesis fills a void, investigating if it is possible to make these environments even more secure without suffering major performance losses.

We have in this thesis introduced two systems that could potentially offer greater secure environments than today's tier one TEEs. The first being a system where the MEE is located before the LLC that is assumed to be shared among multiple cores. By locating the MEE here, all sensitive data and instructions could be encrypted in the LLC and therefore not be read in plain text by other cores. Since the data are encrypted, isolation in the LLC should not be necessary either, as the data is not possible to be comprehended by other cores due to encryption with a key specific belonging to that core. Such a system could solve some attacks such as the Flush+Reload and similar side-channel attacks outlined in Section 2.4, since it exploits a page sharing optimisation in the LLC, where to virtual pages share the same physical page. But if the page of the secure process would be encrypted in LLC, then even if the data is the same but decrypted, physical sharing would not be possible. However, if multiple threads would run on the same core, then they would either have to be assumed to all reside in the same secure environment, or isolation between them would still be necessary for the L2 and L1 caches. The results from the simulations done propose that such a system would not necessarily suffer significant performance losses, as can be seen in Fig. 5.3. But to be noted, the simulations assume that there is no bandwidth limiting the MEE and this parameter should be investigated further.

The second proposed system is to move the MEE even deeper and place it between the core and L1D, only encrypting data and no instructions. Naively moving the

MEE to this location could lead to major performance losses as seen in Fig. 5.1 and Fig. 5.4. One way to reduce this performance loss could be the introduction of a dedicated L0D cache. This would assumably take up a vital area on the chip and therefore a vital point of the analysis was to not only see if it could reduce the performance loss, but also do this with a considerable small-sized L0D. As seen in Fig. 5.5 so were a size of 1 KiB enough in most cases to reduce the performance loss significantly. To reduce it even further, adding more prefetching could be favourable. Such a system should make it impossible for other untrusted threads executing concurrently on the same core to read plain text data. Isolation between threads in such a system should not be needed, following the same argument as for the first system. That same system could even have the potential to battle the Foreshadow attack, introduced in Section 2.4. Even if the attacker could exploit the speculative execution of the processor to read restricted data, (as the Meltdown attack that Foreshadow is based) it would not be able to interpret it since only the secure thread has the right to decrypt the data. However, if only the data would be encrypted, potentially the non-encrypted instructions could be targeted. Moreover, if they would be able to read in plain text a replicating attack could be mounted following the exact instruction flow and branch decisions to then try to extract secret information of the secret environment. One possibility to solve this could be to also keep the instructions encrypted in the L1I and if needed for performance losses, add an L0I cache. The same MEE, (if bandwidth allows it) could then be used for both instruction and data encryption/decryption even in such a system. This has not been investigated in this thesis but would be an interesting deeper study.

Because straightening out if the proposed systems can solve all the exploited weaknesses of modern TEEs are not in the scope of this thesis, we will not discuss the matter further and leave it open. As we have shown in this thesis, integrating the encryption stage of a GPP does not have to mean a significant performance loss. However, if architectural additions are needed for hiding overhead of the MEE, then other factors such as area and power dissipation could be major turning points why not implement such a system. How much area the MEE takes up should be a direct effect of how much bandwidth it is needed to provide and what type of encryption, such as 128, 192 or 256-bit AES. Instinctively, placing it deeper down, where more data is being communicated between memory hierarchy levels a higher bandwidth MEE might be needed. The second system where it is placed before the L1D cache could be needed to have one MEE per core. This does not have to be the case for the first proposed system. One could design a processor where all cores using the same LLC also uses the same MEE, but with a different encryption key. This could have a positive impact on the total area needed. With that said, the MEE's bandwidth impact on performance, the two systems extra occupied area and power dissipated are subjects of interest to investigate further into.

Moreover, with the second system, we investigated the role of hardware prefetching and how it could be used for hiding encryption overheads. There is also a possibility of using software-based prefetching to assist or replace the hardware prefetching. By adding software prefetch instructions in proper time before a read instruction of

encrypted data, performance loss could be limited even further and could work as a supplement or replacement of the hardware prefetching investigated in this thesis.

6.2 Conclusion

To conclude this thesis, we have reviewed how modern secure execution environments work and how they are a target of numerous successful attacks. Most of these work by isolation on-chip and encryption off-chip. We proposed and showed two systems that could potentially solve their vulnerabilities. The first of these systems has its encryption stage moved to before the LLC, the second with it placed before the L1D cache and with an added dedicated cache to the secure environment. They showed from simulations to have minor performance losses compared to a system with no encryption or one with encryption before the main memory, representing a modern system such as Intel SGX or AMD SEV. The second system could also be further improved with hardware prefetching, fetching data to the dedicated cache.

Bibliography

[1] S. Mofrad, F. Zhang, S. Lu, and W. Shi, "A comparison study of intel sgx and amd memory encryption technology," in *Proc. 7th Int. Workshop on Hardware and Architectural Support for Secur. and Privacy*, New York, NY, USA, 2018. [Online]. Available: https://doi.org/10.1145/3214292.3214301

[2] D. Perez-Botero, J. Szefer, and R. B. Lee, "Characterizing hypervisor vulnerabilities in cloud computing servers," in *Proc. 2013 Int. Workshop Secur. Cloud Comput.*, New York, NY, USA, 2013, p. 3–10. [Online]. Available: https://doi.org/10.1145/2484402.2484406

[3] S. Bugiel, S. Nürnberger, T. Pöppelmann, A.-R. Sadeghi, and T. Schneider, "Amazonia: When elasticity snaps back," in *Proc. 18th ACM Conf. Comput. and Commun. Secur.*, New York, NY, USA, 2011, p. 389–400. [Online]. Available: https://doi.org/10.1145/2046707.2046753

[4] J. S. Reuben, "A survey on virtual machine security." Citeseer, 2007. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.626.4718&rep=rep1&type=pdf

[5] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in *Proc. 3rd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst. 2008*, New York, NY, USA, 2008, p. 315–328. [Online]. Available: https://doi.org/10.1145/1352592.1352625

[6] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede, "Hardware-based trusted computing architectures for isolation and attestation," *IEEE Trans. Comput.*, vol. 67, no. 3, pp. 361–374, 2018, doi: 10.1109/TC.2017.2647955.

[7] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Secur. Symp. (USENIX Secur. 16)*, Austin, TX, Aug. 2016, pp. 857–874. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[8] F. McKeen *et al.*, "Innovative instructions and software model for isolated execution," in *Proc. 2nd Int. Workshop Hardware and Architectural Support for Secur. and Privacy*, New York, NY, USA, 2013. [Online]. Available: https://doi.org/10.1145/2487726.2488368

[9] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptol. ePrint Arch.*, 2016. [Online]. Available: https://eprint.iacr.org/2016/086.pdf

[10] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," AMD, White Paper, 2016. [Online]. Available: http://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[11] "Arm security technology building a secure system using trustzone technology," Arm Limited, Cambridge, England, White Paper, April 2009. [Online]. Available: https://developer.arm.com/documentation/PRD29-GENC-009492/c

[12] S. Gueron, "A memory encryption engine suitable for general purpose processors." *IACR Cryptol. ePrint Arch.*, vol. 2016, 2016. [Online]. Available: https://eprint.iacr.org/2016/204.pdf

[13] A. Nilsson, P. N. Bideh, and J. Brorsson, "A survey of published attacks on intel sgx," *arXiv preprint*, 2020. [Online]. Available: https://arxiv.org/abs/2006.13598

[14] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel, "Severed: Subverting amd's virtual machine encryption," in *Proc. 11th Eur. Workshop Syst. Secur.*, New York, NY, USA, 2018. [Online]. Available: https://doi.org/10.1145/3193111.3193112

[15] A. M. Azab *et al.*, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *Proc. 2014 ACM SIGSAC Conf. Comput. and Commun. Secur.*, New York, NY, USA, 2014, p. 90–102. [Online]. Available: https://doi.org/10.1145/2660267.2660350

[16] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How sgx amplifies the power of cache attacks," W. Fischer and N. Homma, Eds., Cham, 2017, pp. 69–90. [Online]. Available: https://doi.org/10.1007/978-3-319-66787-4_4

[17] D. J. Bernstein, "Cache-timing attacks on AES," 2005. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.139.4749&rep=rep1&type=pdf

[18] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx," in *Proc. 10th Eur. Workshop Syst. Secur.*, New York, NY, USA, 2017. [Online]. Available: https://doi.org/10.1145/3065913.3065915

[19] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *Int. Conf. Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Polychronakis and M. Meier, Eds., Cham, 2017, pp. 3–24. [Online]. Available: https://doi.org/10.1007/978-3-319-60876-1_1

[20] J. V. Bulck *et al.*, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Secur. Symp. (USENIX Secur. 18)*. Baltimore, MD: USENIX Association, Aug. 2018, p. 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[21] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symp. Secur. and Privacy (SP)*, 2019, pp. 1–19, doi: 10.1109/SP.2019.00002. [Online]. Available: https://ieeexplore.ieee.org/document/8835233

[22] M. Lipp *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Secur. Symp.m (USENIX Secur. 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[23] M. Schwarz, S. Weiser, and D. Gruss, "Practical enclave malware with intel sgx," in *Int. Conf. Detection of Intrusions and Malware, and Vulnerability Assessment*, R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, Eds., Cham, 2019, pp. 177–196. [Online]. Available: https://doi.org/10.1007/978-3-030-22038-9_9

[24] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[25] "Skylake (server) - microarchitectures - intel," Mar. 2020. [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)

[26] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, p. 1–17, Sep. 2006. [Online]. Available: https://doi.org/10.1145/1186736.1186737

[27] J. L. Hennessy and D. A. Patterson, "Fundamentals of quantitative design and analysis," in *Computer Architecture: A Quantitative Approach*, T. Green and N. McFadden, Eds. 5th ed., Waltham, MA, USA: Elsevier, 2012, ch. 1, pp. 2–71.

[28] "Apple unleashes m1," Nov. 10 2020. [Online]. Available: https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/

[29] T. P. Morgan, "Amping up the arm server roadmap," Dec. 13 2019. [Online]. Available: https://www.nextplatform.com/2019/12/13/amping-up-the-arm-server-roadmap/

[30] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, Aug 2014. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html

[31] AMD, "Amd epyc™ 7002 series processors: A new standard for the modern data center," Apr. 2020. [Online]. Available: https://www.amd.com/system/files/documents/AMD-EPYC-7002-Series-Datasheet.pdf

[32] J. L. Hennessy and D. A. Patterson, "Instruction-level parallelism and its exploitation," in *Computer Architecture: A Quantitative Approach*, T. Green and N. McFadden, Eds. 5th ed., Waltham, MA, USA: Elsevier, 2012, ch. 3, pp. 148–261.

[33] ARM, "Arm cortex-m7 processor datasheet," 2020. [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m7

[34] A. Matta, "Arm neoverse v1 platform: Unleashing a new performance tier for arm-based computing," Arm Community, Apr. 17 2021. [Online]. Available: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/neoverse-v1-platform-a-new-performance-tier-for-arm?_ga=2.149537279.856418819.1620289544-65415261.1613143471

[35] M. Kerner and N. Padgett, "A history of modern 64-bit computing," Feb. 2007. [Online]. Available: https://courses.cs.washington.edu/courses/csep590a/06au/projects/history-64-bit.pdf

[36] J. L. Hennessy and D. A. Patterson, "Memory hierarchy design," in *Computer Architecture: A Quantitative Approach*, T. Green and N. McFadden, Eds. Waltham, MA, USA: Elsevier, 2012, ch. 2, pp. 72–147.

[37] S. Mittal, "A survey of recent prefetching techniques for processor caches," *ACM Comput. Surv.*, vol. 49, no. 2, Aug. 2016. [Online]. Available: https://doi.org/10.1145/2907071

[38] F. Zhang and H. Zhang, "Sok: A study of using hardware-assisted isolated execution environments for security," in *Proc. Hardware and Arch. Support for Sec. and Privacy 2016*, New York, NY, USA, 2016. [Online]. Available: https://doi.org/10.1145/2948618.2948621

[39] S. Johnson, R. Makaram, A. Santoni, and V. Scarlata, "Supporting intel sgx on multi-socket platforms," Intel Corporation. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions/supporting-sgx-on-multi-socket-platforms.html

[40] *Advanced Encryption Standard*, FIPS PUB 197, Nov. 26 2001. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[41] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *J. of Cryptogr. Eng.*, vol. 8, no. 1, pp. 1–27, 2018. [Online]. Available: https://doi.org/10.1007/s13389-016-0141-6

[42] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *J. Cryptogr. Eng.*, vol. 1, no. 1, pp. 5–27, 2011. [Online]. Available: https://doi.org/10.1007/s13389-011-0006-y

[43] J.-J. Quisquater and D. Samyde, "Electromagnetic analysis (ema): Measures and counter-measures for smart cards," in *Int. Conf. Res. in Smart Cards*, 2001, pp. 200–210. [Online]. Available: https://doi.org/10.1007/3-540-45418-7_17

[44] D. Genkin, L. Pachmanov, I. Pipman, A. Shamir, and E. Tromer, "Physical key extraction attacks on pcs," *Commun. ACM*, vol. 59, no. 6, pp. 70–79, 2016, doi: 10.1145/2851486.

[45] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Topics in Cryptology – CT-RSA 2006*, D. Pointcheval, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20. [Online]. Available: https://doi.org/10.1007/11605805_1

[46] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Secur. Symp. (USENIX Secur. 14)*, San Diego, CA, USA, Aug. 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[47] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symp. Secur. and Privacy*, 2015, pp. 640–656, doi: 10.1109/SP.2015.45. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7163052

[48] J. Van Bulck, F. Piessens, and R. Strackx, "Sgx-step: A practical attack framework for precise enclave execution control," in *Proc. 2nd Workshop Syst. Softw. for Trusted Exec.*, New York, NY, USA, 2017. [Online]. Available: https://doi.org/10.1145/3152701.3152706

[49] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *2019 IEEE Eur. Symp. Secur. and Privacy (EuroS P)*, 2019, pp. 142–157, doi: 10.1109/EuroSP.2019.00020. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8806740

[50] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop Offensive Technol. (WOOT 18)*, Baltimore, MD, USA, Aug. 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[51] "Skylake processors," High-End Computing Capability, May 13 2021. [Online]. Available: https://www.nas.nasa.gov/hecc/support/kb/skylake-processors_550.html

[52] K. Hoste, "SPEC CPU2006 command lines," Sept. 19 2012. [Online]. Available: http://www.cs.ucy.ac.cy/courses/EPL605/Fall2014Files/SPEC%20CPU2006%20command%20lines.pdf

[53] Y. Luo, L. John, and L. Eeckhout, "Self-monitored adaptive cache warm-up for microprocessor simulation," in *16th Symp. Comp. Arch. and High Perf. Comput.*, 2004, pp. 10–17.

[54] T. Lundqvist and P. Stenstrom, "Timing anomalies in dynamically scheduled microprocessors," in *Proc. 20th IEEE Real-Time Sys. Symp. (Cat. No.99CB37054)*, 1999, pp. 12–21, doi: 10.1109/REAL.1999.818824.

A

Appendix 1

A.1 Analysing simulations dependency on simulated instructions

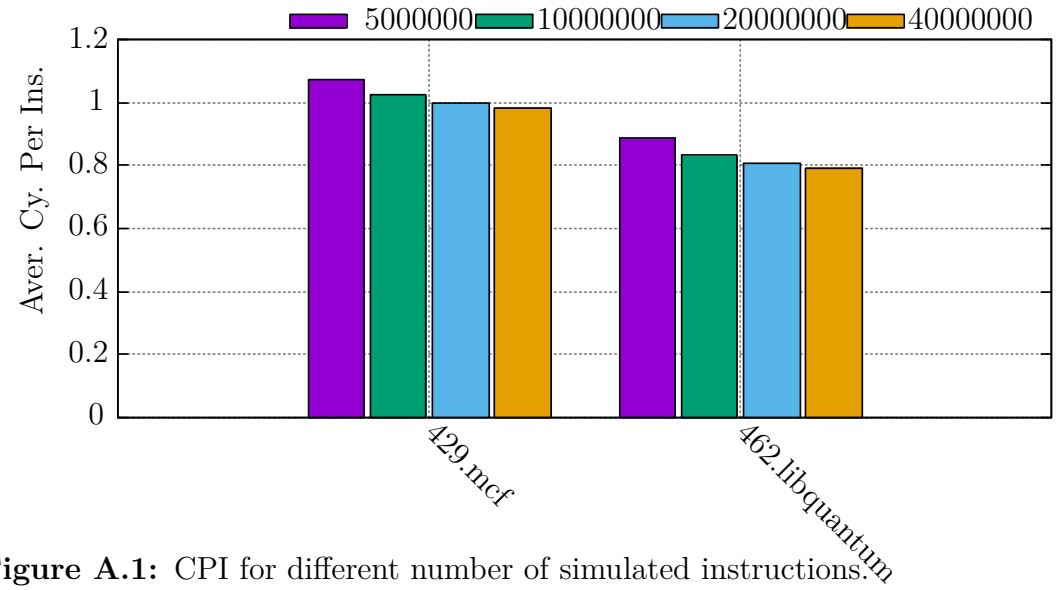


Figure A.1: CPI for different number of simulated instructions.

Fig. A.1 shows an analysis of how the number of simulated instructions is affecting the Clocks Per Instruction (CPI) parameter. It can be seen that it is steadily decreasing but flattening out with higher instruction counts which are expected. The big difference between 5000000 and 40000000 instructions shows the importance of running a warm-up of the cache to lower the impact of filling up caches at the beginning of executing a new program.

A.2 Compiling and Running Benchmarks

The benchmarks were compiled on a x86-64 Ubuntu 18.04 system with the compiling flags:

COPTIMIZE =
CXXOPTIMIZE = −std=c++98 –O2
FOPTIMIZE = –O2

For the compilers GCC, G++ and GNU Fortran, the version used were Ubuntu 7.5.0-3ubuntu1~18.04.

As long as a benchmark did not give compile error, run error on desktop or error running on gem5 it was assumed to be working. The benchmark was tried running for about 10 seconds on the desktop before aborted, and for minimum 50 000 000 instructions on the gem5 simulator, using the atomic CPU. Comments for each benchmark can be found in Tab. A.1.

Benchmark	Working
400.perlbench	No output desktop/gem5.
401.bzip2	Output desktop/gem5, gem5 gives just the few first lines. output after a while, changing from input.source to chicken.jpg will give more output.
403.gcc	No output desktop/gem5.
410.bwaves	Same output desktop/gem5.
416.gamess	Same output desktop/gem5.
429.mcf	Same output desktop/gem5.
433.milc	Same output desktop/gem5.
435.gromacs	No output desktop/gem5.
436.cactusADM	Same output desktop/gem5.
437.leslie3d	No output desktop/gem5.
444.namd	Output for desktop but no produced for gem5. Ran for 10^8 instructions with no error.
445.gobmk	Output on desktop, about 42 000 000 instructions finish simulation, but still no output on gem5.
447.deaII	Error compiling
450.soplex	Same output desktop/gem5. After 22 000 000 instructions, warnings with remapping to new vaddr.
453.povray	Same output desktop/gem5.
454.caclulix	Output on desktop, gem5 gives same intro text but then no more.
456.hmmer	Same output desktop/gem5.
458.sjeng	Same output desktop/gem5.
459.GemsFDTD	Same output desktop/gem5.
462.libquantum	Same output desktop/gem5.
464.h264ref	Same output desktop/gem5.
465.tonot	No output desktop/gem5.
470.lbm	Same output desktop/gem5.
471.omnetpp	Output on desktop, gem5 gives same intro text but then no more.
473.astar	Same output desktop/gem5.
481.wrf	Error running on desktop
482.sphinx3	Same output desktop/gem5.
483.xalancbmk	Error compiling.
998.specrand	Same output desktop/gem5.
999.specrand	Same output desktop/gem5.

Table A.1: Benchmark simulation comments.

A.3 Full Configuration of Base System

CPU	7 stage pipline simulated for minimum 14 cycles (14 stages)
Fetch	fetchToDecodeDelay=2, fetchWidth=8, fetchBufferSize=64, fetchQueueSize=32
Decode	decodeToRenameDelay=2, decodeWidth=8, decodeToFetchDelay=1
Rename	renameToIEWDelay=4, renameWidth=8, renameToFetchDelay=1, renameToDecodeDelay=1
IEW	issueToExecuteDelay=2, iewToCommitDelay=2, issueWidth=8, dispatchWidth=8, wbWidth=8 iewToFetchDelay=1, iewToDecodeDelay=1, iewToRenameDelay=1
Functional units	6 IntALU, 2 IntMultDiv, 4 FP_ALU, 2 FP_MultDiv 4 SIMD_UNIT, 1 PredALU, 4 RdWrPortm 1 IprPort
Commit	CommitToFetchDelay=1, commitToDecodeDelay=1, commitToRenameDelay=1, commitToIEWDelay=1, commitWidth=8, squashWidth=8,
ROB	numRobs=1, numROBEntries=224, renameToROBDelay = 1,
Registers	numPhysIntRegs=256, numPhysFloatRegs=256, numPhysCCRegs=numPhysIntRegs*5, numPhysVecRegs=256, numPhysVecPredRegs=32
SMT	smtNumFetchingThreads=1, smtFetchPolixt=SingleThread, smtLSQPolicy=Partitioned, smtLSQThreshold=100, smtIQPolicy=Partitioned, smtIQThreshold=100, smtROBPolicy=Partitioned, smtROBThreshold=100,
Queues	LQEntries=32, SQEntries=32, LSQDepCheckShift=4, LSQCheckLoads=True, numIQEntries=64,
Others	trapLatency=13, fetchTrapLatency=1, backComSize=5, forwardComSize=5, store_set_clear_period=250000, LFSTSize=1024, SSITSize=1024

Table A.2: CPU setup for base system simulated.

Non-unique values
warmup_percentage=0, max_miss_count=0, demand_mshr_reserve=1, tgts_per_mshr=12, tags=BaseSetAssoc(), tags=BaseSetAssoc(), replacement_policy=LRURP(), sequential_access=False, writeback_clean=False
L3 shared, size is set per core
size=2 MiB, assoc=11, mostly_inclusive, tag_latency=8, data_latency=50, response_latency=50, mshrs=40, write_buffers=32, prefetcher=NULL, prefetch_on_access=False
L2 private per core
size=1 MiB, assoc=16, mostly_inclusive, tag_latency=4, data_latency=14, response_latency=14, mshrs=20, write_buffers=16, prefetcher=StridePrefetcher(), prefetch_on_access=True
L1 split in half between I/D
size=64 KiB, assoc=8, mostly_inclusive, tag_latency=2, data_latency=4, response_latency=4, mshrs=10, write_buffers=8 prefetcher=NULL, prefetch_on_access=False
System DRAM (Main Memory)
latency 55 ns = 205 cycles at 3.2 GHz, bandwidth = 12.8 GB/s

Table A.3: Memory setup for base system simulated.