Kurt Nistelberger

CacheHammer - A Software-based Fault Attack

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor: Michael Schwarz

Institute for Applied Information Processing and Communication

Graz, February 2020

AFFIDAVIT

EIDESSTATTLICHE ERKLÄRUNG

Abstract

Power consumption is an important criteria for the selection of a CPU. The power is mostly determined by two variables, the supplied voltage and frequency. Keeping these knobs low leads to a strict reduction of power consumption and also keeps heat production in the processor low. This goal led to a rapid development and optimization in the power management of CPUs. CPUs are not constantly operating at their highest peak, therefore it is not always the need to supply the maximum power to a CPU core. Adjusting the power supply to the current workload leads to several performance optimizations. On Intel CPUs, this is handled via performance states that define different voltage-frequency pairs which are applied depending on the current need. The rapid change of supply domains can also introduce security risks as it has been shown in the past on ARM CPUs. Reducing frequency and voltage to a critical low level can introduce faulty computation results in the CPU core. These wrong calculations could later be used to leak secret AES keys from a trusted execution environment on a mobile phone.

In this thesis, we extend the approach of attacking mobile phones to regular desktop PCs and verified that it is possible to run desktop CPUs on unstable operating points and induce faults. This can be achieved by the usage of an undocumented Model Specific Register (MSR) which allows to define the undervolt level of voltage planes in the CPU. We stress tested the instruction cache and also several execution units and analyzed their behavior on specific critical operating points. With our test cases we could show that certain Intel CPUs are more prone to fault injections than others. Using our results it is possible to mount targeted attacks against cryptographic implementations which are running inside Intel's Software Guard Extension. We also show that we are able to stabilize our attacks by using fast frequency scaling next to undervolting the CPU. This reduces the chance for a system crash and leads to more possibilities for future attacks using this vector.

With this work, we showed that a reduction in the power supply of a CPU can lead to unexpected computation results. Furthermore, these faulty calculations can be used to target cryptography, showing that energy-saving options can have catastrophic effects.

Kurzfassung

Energieverbrauch ist eine wichtige Eigenschaft bei der Auswahl einer CPU. Die Leistung wird hauptsächlich von zwei Variablen definiert, der Versorgungsspannung und der Frequenz. Der Leistungsverbrauch wird stark reduziert wenn beide Werte dieser Veränderlichen klein gehalten werden und somit bleibt auch die damit verbunde Wärmeentwicklung klein. Dieses Ziel führte zu einer rasanten Entwicklung und Optimierung der Energieversorgung von CPUs. CPUs arbeiten nicht durchgehend auf ihrem vollen Potential, deshalb ist auch nicht immer die volle Versorgung nötig. Eine variable Spannung angepasst an die aktuelle Arbeit hat zu einigen Leistungsoptimierungen geführt. Auf Intel CPUs wird dies über so genannte Leistungs-Punkte geregelt, diese Punkte definieren verschiedene Spannungs- und Frequenzwerte welche gemeinsam angewendet werden können. Dass die schnelle Entwicklung in den Versorgungselementen kann auch Sicherheitsrisken mit sich bringen kann wurde in der Vergangenheit an ARM CPUs gezeigt. Eine Reduktion der Frequenz und der Spannungs auf ein kritisch minimales Level kann Fehler in Berechungen zur Folge haben. Diese falschen Ergebnisse konnten benutzt werden um geheime AES Schlüssel aus vertrauten Umgebungen auf mobilen Endgeräten zu extrahieren.

Diese Arbeit zielt darauf ab, dass Angriffszenario von mobilen Endgeräten auf konventionelle Desktop PCs zu erweitern. Wir konnten zeigen und verifizieren, dass es möglich ist Desktop PCs auf instabilen Arbeitspunkten arbeiten zu lassen und Fehler zu produzieren. Dies kann erreicht werden durch die Benützung von undokumentierten Model Specific Register (MSR) welche es erlauben die Versorgungsspannung der CPU zu regulieren. Wir haben den Instruction Cache sowie auch mehrere Execution Units getested und deren Verhalten bei kritischen Arbeitspunkten analysiert. Mit unseren Ergebnissen ist es möglich gezielte Angriffe gegen kryptographische Implementierungen durchzuführen welche inerhalb von Intel's Software Guard Extension laufen. Weiters zeigten wir auch das falsche Ergebnisse leichter produziert werden können wenn zusätzlich zur verminderten Spannung die Frequenz alternierend geändert wird.

Mit dieser Arbeit haben wir gezeigt das eine Reduzierung der Spannung von CPUs zu unerwarteten Berechnungsergebnissen führen kann. Weiters wird gezeigt das diese Fehlberechnungen ausgenützt werden und zu katastrophalen Effekten führen können. Dies sollte beachtet werden bei jedem Benutzer der die Spannung reduziert um damit Energie zu sparen.

Acknowledgments

First of all, I want to thank my advisor Michael Schwarz for his exceptional support and patients. All the effort he put into answering my questions, the helpful discussions and the reviews of this thesis.

I also want to thank my parents and my brothers, for the never-ending support throughout my studies and during the work on this thesis.

Kurt Nistelberger

Contents

Chapter 1

Introduction

The continuous performance increase of Central Processing Units (CPU) comes with an increase in power consumption as well. A comparison between the first microprocessor that consumed less than a watt and the latest Intel Core i9-9900K consumes 95W [36] shows a tremendous increase. Considering that modern chips have an average length of around 1.5cm, the extra heat which is produced by the consumption of electrical energy has to be dissipated from the chip. The research is close to the possible limit and stuck on this problem for several years [55]. To overcome this issue, different solutions can be applied from simple countermeasures like shutting down unused units to more complex one in reducing the supply voltage and scaling of the base frequency [68].

Decreasing the supply voltage on elements of a computer has been shown to be very dangerous and must be done carefully. Due to low voltages and smaller design constraints, accesses to memory locations in a sufficiently high frequency caused bitflips in neighboring cells. This is called the Rowhammer effect [23, 41, 43]. Exploiting this technique allows breaking memory isolation and also accessing bits from other processes, which would not be possible otherwise. Abusing this behavior can lead to privilege escalation attacks on almost all systems. This security vulnerability has been tried to be mitigated in several attempts [2, 11, 56, 77]. But the main memory can somehow be always altered, for example, by privileged code or by the operating system itself. Therefore, Intel has considered the main memory as an untrusted storage source and introduced an encrypted and authenticated private memory region. This new trusted hardware unit in Intel CPUs is called Software Guard Extension (SGX). The private memory region used by SGX is called an enclave, and only code from the associated enclave can alter the used memory region. Every change from outside of the enclave is noticed during decryption and leads to a system halt. This technique provides a secure execution environment where not even the operating system has access to the memory used by SGX [24]. This trusted storage has the potential to solve many security and privacy challenges, for example, secure remote communication or digital rights management [44].

Since the introduction of SGX, it has been frequently targeted and tried to be exploited by researchers [71, 25, 10]. The amount of attacks shows that it is not a simple task to guarantee a secure and trusted execution environment.

1.1 Motivation for the Thesis

Cache memory is a small and fast storage located between the place where instructions get executed and the main memory to speed up the execution time. The Rowhammer effect attacks the main memory, and this gave us the idea and motivation to verify if the cache memory is vulnerable to the same effect. Bit flips in the cache memory can break the SGX secure enclave.

The encryption and authentication operations from SGX are performed in the Memory Encryption Engine (MEE), this unit is located after the cache hierarchy and is not protecting data which is stored in the CPU caches. Therefore, a bit flip in the caches is not detected by SGX [13]. This leads to a violation of the SGX security guarantee and possible new attacks on the trusted environment.

A motivating example that our idea might work has been shown by the work of Tang et al. [69]. Tang et al. abused low voltage levels to break trusted execution environments on the ARM architecture.

1.2 Contribution

Our contribution to the field of attacks raised by bit flips is the analysis of how the instruction cache is behaving during undervolting. As well as how we can achieve a high access rate on the instruction cache to increase the likelihood of bit flips.

Our analysis showed that it is possible to trigger bit flips during code execution in an undervolted environment. Against our assumption of producing flips in the instruction cache, we achieved bit flips in the execution units of the CPU. This is as critical as flips in the cache, and we managed to exploit this flaw and targeted cryptographic algorithms.

Chapter 2

Background

In this chapter, we discuss in detail the background information required to understand the outcome of this thesis. We define several acronyms and terms which are used in later chapters. In Section 2.1, we give an overview of the Intel CPU architecture. In Section 2.2, we explain the internals of CPU caches. Section 2.3, describes further cache levels until the final execution of an instruction. In Section 2.4, we go into detail of the inner construction of caches, describing a single storage element. Section 2.5 discusses the process of instruction fetching until the execution of decoded instructions. In Section 2.6, we explain the power management of CPUs and technology to reduce the overall power consumption. Furthermore, in Section 2.7, the Intel Software Guard Extension is explained, and in the last Section 2.8, an overview of asymmetric cryptography is given.

2.1 Intel CPU Architecture

A CPU is executing instructions of a computer program by performing basic arithmetic, logic, and control operations which are defined by the CPUs *instruction set architecture* (ISA). The ISA of a CPU defines which instructions are accepted and can be executed and which are not. By defining the term CPU architecture, it needs to be distinguished between the ISA and the *microarchitecture*. The programmer's visible code is referred to the ISA. It can be seen as a boundary between the software and hardware [55]. In essence, it can be said that the microarchitecture is the hardware implementation of the ISA. For example, processors from AMD and Intel might have the same ISA but an utterly different microarchitecture like Zen and Skylake [55]. Since the ISA is the same, allowing a program to run on both CPUs.

The Intel Architecture is based on a complex instruction set computer (CISC) scheme. Compared to a reduced instruction set computer (RISC), it has more complex instructions implemented and allows to produce a program with fewer assembly instructions since one instruction is able to perform more steps [57]. This has the disadvantage that single instruction takes longer to execute compared to the simpler instructions from a RISC CPU.

The following explanations are based on an Intel 6th generation Skylake CPU. The tests in this thesis are performed on a Skylake CPU i5-6200U and a Kabylake CPU i3-7100U. Their internal architecture is assumed to be very similar.

The microarchitecture of an Intel CPU, as well as other modern CPUs, can be simplified into three regions the front-end, execution units, and the memory subsystem. The front-end is responsible for the instruction fetch and decoding. In the execution units the decoded instructions are getting executed out-of-order. The memory subsystem contains the three levels of caches and performs store and load operations [53].

The front-end is fetching instructions from the instruction cache and decoding it into micro operations. The decoded instructions are also called *μops* and represent an intermediate state of an instruction before it is forwarded to the execution units [60]. The decoding of the instructions is done in the hardware part called decoder. The decoder in a Skylake architecture consists of four simple decoder units and one complex decoder unit which are implemented in hardware. For more complex instructions which are decoded into multiple *μops*, the microcode sequencer ROM (MS ROM) is used [53]. The sequencer contains a read-only lookup table where the *μops* for specific instructions are stored. The sequencer also contains a writeable memory region for microcode updates. This is the region were bugs like the Intel Pentium fdiv bug get fixed [12]. This is a critical part of the Intel CPU and very well protected. If an attacker can install an arbitrary microcode update, the attacker is able to gain full control by modifying the behavior of instructions [42, 9].
Furthermore, if a program is only using a subset of itself, for example, it is executing a loop. The front-end may already have stored the decoded instructions in the decoded stream buffer (DSB). This increases the performance of the whole system since instructions do not have to be decoded. A further performance improvement of Intel CPUs is called Loop Stream Detector (LSD). An LSD can store loops that contain less than 64 decoded instructions and directly deliver them to the execution units. If *μops* are not stored in the DSB and cannot be delivered from the LSD, for example, by calling a function or jumping to another region, the front-end fetches the instruction from the level 1 cache and decodes it. This is also called the legacy path [75].

In contrast to the front-end, the back-end is operating out-of-order and also applying several optimizations to gain faster execution of the *μops*. Before *μops* are getting executed, they are stored in the reorder buffer (ROB). The ROB holds the *μops* during the time of execution to the time of commit. The scheduler becomes *μops* from the ROB and forwards them to the corresponding execution unit where they get finally executed. After successful execution, the result is returned to the ROB and gets committed to the register file. In case an exception happens during execution, the result is thrown away

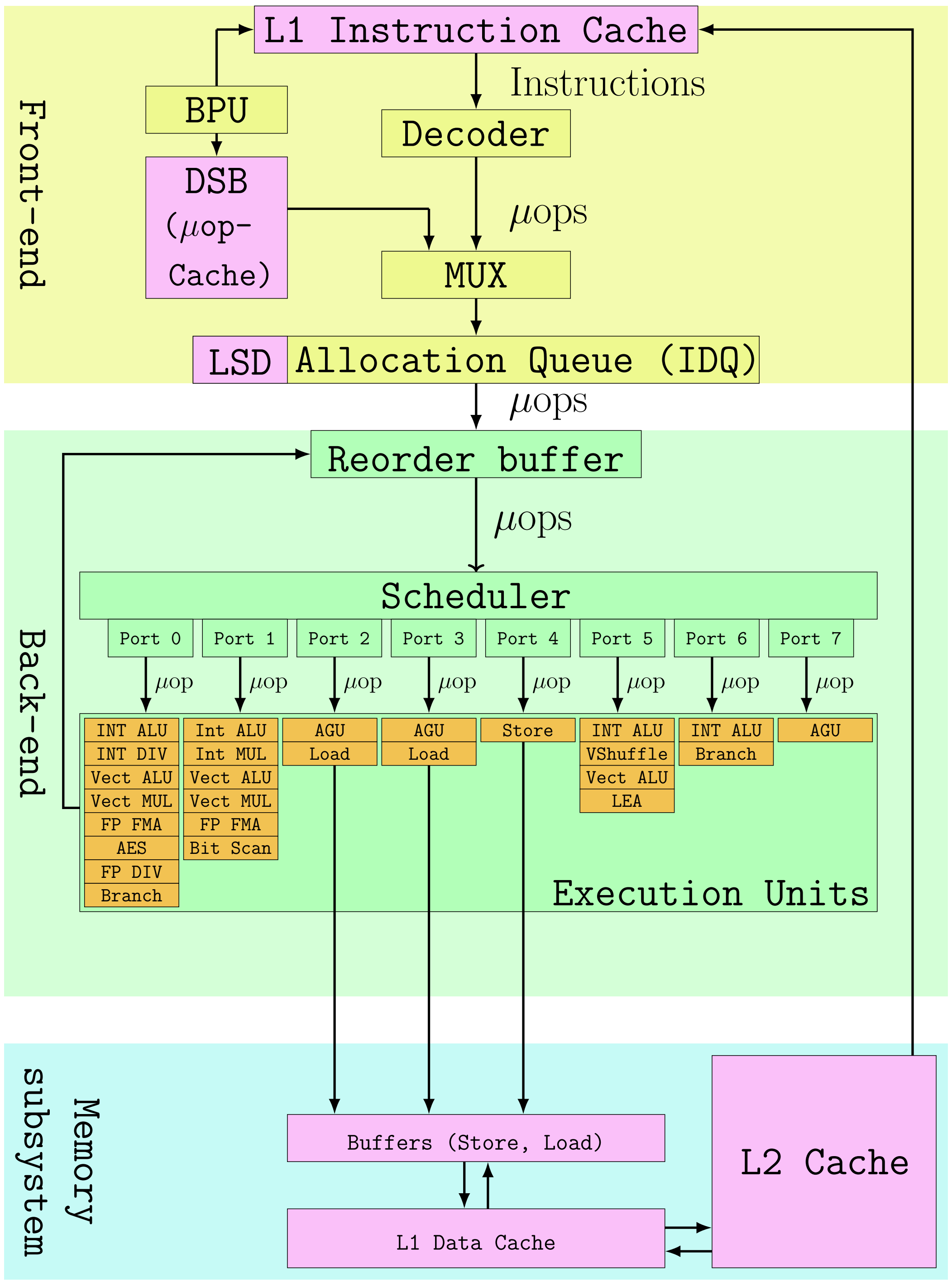

Figure 2.1: Simplified depiction of an Intel Kaby Lake CPU, which contains the three main regions front-end, back-end, and memory subsystem. Instructions are fetched from the instruction cache and passed through the front-end, get decoded into *µops*, and forwarded to the back-end, after execution they return to the reorder buffer where they might retire. The memory subsystem handles loads and stores.

and not committed [55]. The Skylake architecture has eight different execution ports, for better balancing and delegation of the tasks [75]. The back-end does not guarantee that all operations are executed in-order. If two *μops* o_1 and o_2 are issued in-order, this does not mean that they are executed in the same order, and it might happen that o_2 finishes before o_1. To keep track of the out-of-order execution, the operations are retired in-order.

A high level overview of the Intel Skylake architecture is shown in Figure 2.1. These resources are available per physical core. If the CPU supports Hyper Threading (HT), two hardware threads run on the same core, sharing the resources. This optimization exploits the thread-level parallelism (TLP) technology, which supports multiple threads without requiring an intervening process switch [55]. This fast switching between threads is used to hide the latency introduced by pipeline and memory. HT provides better performance per watt than increasing the frequency and is now supported by almost every CPU [53]. Two threads on the same physical core have their own register file, but the memory is shared. The size of the *μop*-Cache is not shared it is equally partitioned between hyper threads [31].

2.2 CPU Cache

To execute a program, the CPU has to fetch the instructions and data needed from the main memory (DRAM). To speed up the data transfer between the DRAM and the CPU an intermediate memory layer exists called cache. The CPU can store data in registers, but those are very limited in size, and therefore the cache exists. A cache contains temporary copies of data in the main memory, which is likely to be used by the processor sooner or later. This is possible because code and data have spatial and temporal locality [55]. Caches are small and fast memory designed to bridge the gap between the CPU and the main memory [18].

To avoid a high delay from memory accesses, caches have been optimized a lot in recent years for better performance and speed. A typical cache has a hierarchical structure with multiple levels. Cache levels with small size are faster in access time than bigger cache levels due to addressing overhead [49]. Whenever the CPU needs data, it first asks the cache, and if the cache does not have it stored (cache miss), it requests it from the main memory. Caches are limited in size. Therefore, the main memory is still needed. Increasing the cache size would lead to slower access times and make the cache less useful. The cache has to fulfill a size-speed trade-off.

2.2.1 Cache Structure

Modern CPUs have a hierarchically structured cache [55], usually consisting of 3 levels called L1, L2, and L3. The level 3 cache is also called last level cache (LLC). The first level cache (L1) is the smallest and the fastest and closest to the CPU core, it is split into an instruction part (L1I) and a data part (L1D) to gain even further speed. The hierarchy of the cache levels is shown in Figure 2.2.

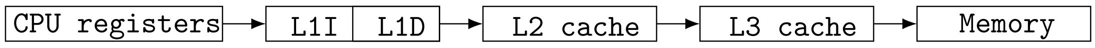

Figure 2.2: Memory hierarchy.

CPUs usually contain multiple cores. Each core has a private L1 and L2 cache. The LLC is typically shared between cores. Because of the private L1 and L2 caches, the data in the cores might get out of sync. To keep the data consistent, cache coherency protocols are required to update the data or to invalidate it if necessary [55].

There are several ways to design multi-level caches. One approach is to store all blocks present in L1 cache also in the L2 and everything from the L2 cache also in the LLC. This scheme is called inclusive, the data is stored multiple times, which is a waste of space, but the update mechanism between the cache levels is simple. The significant advantage of this method is that only the first level cache accesses need to be tracked. The opposite approach is that data is not stored multiple times across cache levels. This inevitably makes the synchronization mechanism more complex. This approach is called an exclusive cache. If the scheme is neither strictly inclusive nor strictly exclusive, it is a mixed scheme and called non-inclusive non-exclusive [55].

Data is transferred between memory and cache lines in blocks of fixed sizes. Cache lines are grouped in a cache sets. A modern cache is divided into several sets. This structure is called a set-associative mapping and can be found in almost all modern CPUs. Compared to a direct-mapped cache where data from one specific address can be stored at exactly one location in the cache, the set-associative structure comes with more flexibility. Fully-associative caches have the highest flexibility, but they come with an extra lookup table to store the information in which data is stored in which cache line [26]. The inner structure of a set-associative cache can be seen in Figure 2.3.

All test cases in this thesis where done on two CPUs, an i3-7100U and an i5-6200U. The cache sizes of both CPUs are shown in Table 2.1.

Table 2.1: Caches sizes and associativity of our two test CPUs [75].

CPU	L1	L2	L3
i3-7100U	2x32KiB 8-way set assoc.	256KiB 4-way set assoc.	2MiB Up to 16-way set assoc.
i5-6200U	2x32KiB 8-way set assoc.	256KiB 4-way set assoc.	2MiB Up to 16-way set assoc.

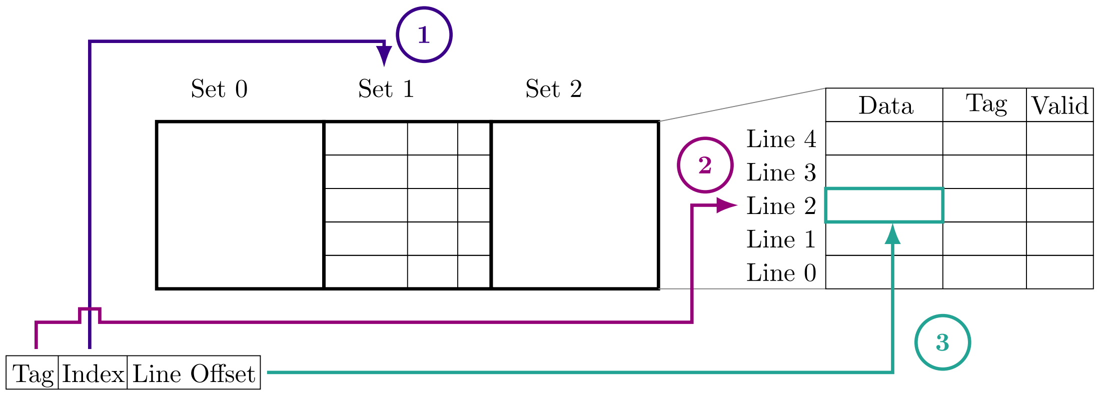

Figure 2.3: Scheme of a 5-way 3-set associative cache. From the address index, tag and cache line offset can be retrieved. These are used to select the correct data. The index selects the cache set(1), the tag refers to the cache line in the selected set(2), and the offset determines from which byte the data fetch starts(3). The valid bit has to be true in order to retrieve the data.

2.3 Instruction Caches

For better understanding the results of this thesis, a more in-depth look into caches is necessary. As explained in 2.2, the level 1 cache is split into an instruction part and a data part. The instruction cache is responsible for delivering instructions to the CPU. This throughput is crucial for the performance of a CPU [31]. To gain better instruction throughput, additional layers are introduced between the L1I and the execution units.

2.3.1 Decoded Stream Buffer (DSB)

Between L1I and the execution units resides another cache for already decoded instructions. This cache has several names, such as Decoded Stream Buffer (DSB), decoded ICache, or *μop*-Cache. For consistency, we call it *μop*-Cache for the rest of the thesis. This extra level of cache deals better with the principle of locality by storing already decoded instructions closer to the execution units. Allowing the CPU to fetch instructions faster since it does not have to wait for the instructions to get decoded by the pipeline [31]. The *μop*-Cache can be seen in figure 2.1 located in the front-end.

The *μop*-Cache on an Intel Skylake and Kabylake CPU can hold up to 1.536 micro instructions in a 32 set 8-way set associative cache, leading to 6 *μops* per cache line. The *μop*-Cache is statically divided between threads on a core and inclusive with the L1 instruction cache.

The *μop*-Cache is an accelerator of the legacy pipeline. By storing decoded instructions, it enables the following features [31].

- reduced latency on branch mispredictions
- increased *μop* delivery bandwidth to execution units
- reduced front end power consumption

The *μop*-Cache stores the output of the instruction decoder. Next time a micro-op is consumed for execution, the decoded instructions are taken from the *μop*-Cache. This allows skipping the legacy pipeline and reduces the power and latency of the front-end. The *μop*-Cache provides an average hit rate of above 80%. It performs even better on program parts called "hot spots". These are operation loops without any further branches that fit in the *μop*-Cache. Hot spots can have a hit rate close to 100% [31].

To use the *μop*-Cache optimally and fully exploit its potential, there are several rules which define if a decoded instruction is stored in the cache. The most important are listed below.

- *μops* in a way are representing statically contiguous instructions
- instructions which get decoded into multiple *μops* cannot be split across ways
- a non-conditional branch is the last micro-op in a way

When these restrictions are fulfilled, *μops* are fetched from the *μop*-Cache. If they cannot be stored, they are delivered from the decode pipeline. Switching back to the *μop*-Cache can only occur after a branch instruction. Frequent switches between the pipeline and the *μop*-Cache can cause extra timing penalty [31].

The *μop*-Cache is included in the instruction cache and the ITLB. This means that if instructions are removed from the L1I using eviction, they are also evicted from the *μop*-Cache. An ITLB entry eviction is even causing a flush of the entire *μop*-Cache.

2.3.2 Loop Stream Detector (LSD)

The LSD is an additional optimization within the Instruction Decode Queue (IDQ). It is capable of detecting loops which fit in the IDQ and locks them down. The LSD can continuously provide *μops* without fetching and decoding instructions or accessing any other cache. This is a continuous stream until a branch misprediction occurs. The LSD is able to detect loops in the size of up to 64 *μops* per thread. As for the *μop*-Cache as well for the LSD, the code residing in the LSD has to fulfill requirements [31].

- all *μops* have to be also stored in the *μop*-Cache
- can contain up to eight taken branches, but no CALL or RET instructions
- stack operations (PUSH, POP) have to match

These requirements are fulfilled by many calculation-intensive loops like string copying or modular exponentiation. It is worth to mention that the Intel Optimization Manual states that loop unrolling should be preferred even when it is overflowing the LSD capability [31].

While the loop stream detector is delivering instructions, the rest of the front-end in a core is effectively disabled, leading to an increased power save.

2.4 Static Random Access Memory

Static Random Access Memory (SRAM) is a widely used semiconductor memory technique in electronics and used whenever fast and low energy memory is needed. The name stems from the static data storage technology, by using a latch of two cross-coupled inverters. In contrast to DRAM, the data in an SRAM cell does not have to be refreshed, but it is still volatile and the data is lost when the power is removed [15].

2.4.1 6T SRAM Cell

Compared to *Dynamic Random Access Memory* (DRAM), SRAM cells do not have to be refreshed. This speeds up the access time, which can be held close to the cycle time [55]. One SRAM cell contains six transistors. Four are configured as two cross-coupled inverters to keep the inner state one as write gate and one as read gate. Therefore, the cell is called 6T SRAM cell [15, 55].

The operation mode of SRAM has several advantages compared to DRAM, especially in speed and power consumption. In idle mode, it consumes only a fraction of the power, and it is easier to control since there are no refresh cycles needed. The disadvantages of SRAM is that they are more significant in die size and more expensive than DRAM. DRAM cells are used in main memory, SRAM cells are preferably used in caches. Cache levels have different sizes leading to 2-8 times slower access time [49] of the last level cache compared to the second-level cache due to the extra addressing overhead. Although the LLC is still five times faster than a DRAM access [15, 55]. Comparing the energy consumption shows that SRAM access is 125 times more energy-efficient than a DRAM access [27], making it even more interesting for high-performance systems.

2.4.2 Read and write stability

6T SRAM cells are designed to hold data during read accesses and change quickly on a write access. This is a conflicting requirement and an important condition an SRAM cell has to fulfill. Increasing power savings of CPUs forces the SRAM memory to operate on variadic input voltage but still achieving the requirement of read and write stability. Sangeeta et al. [20] state that reading an SRAM cell is a critical operation. Too low operating voltages might lead to a change in the stored value during a cell read. Gadhe

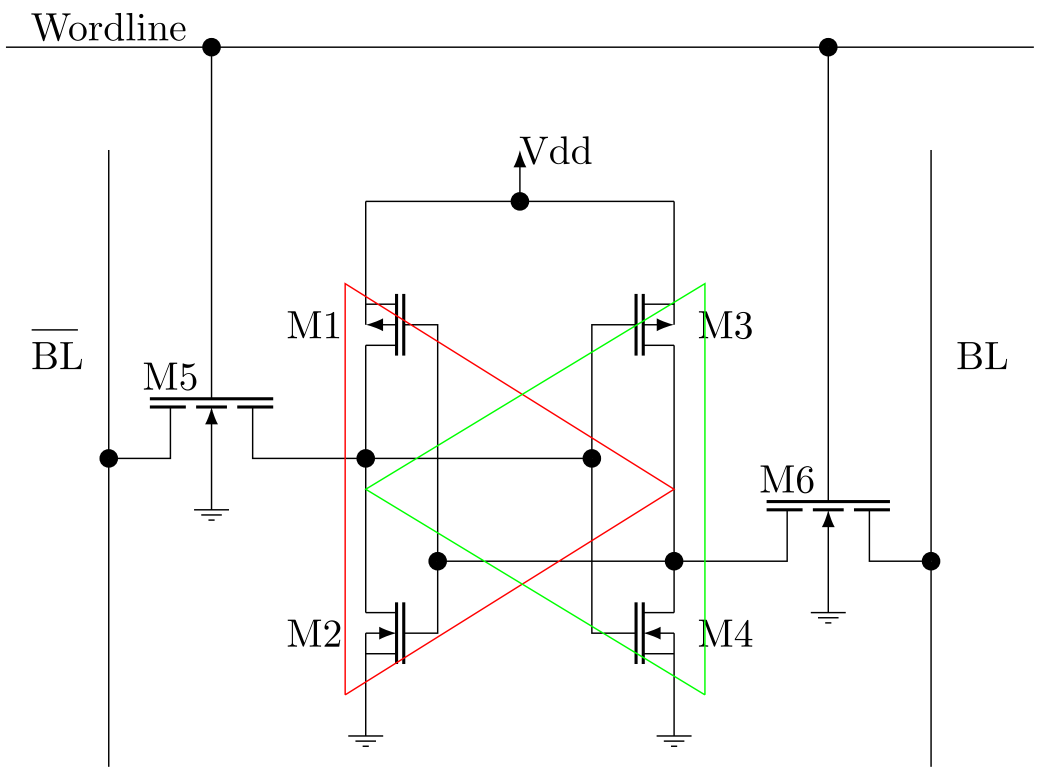

Figure 2.4: Circuit of a six transistor SRAM cell consisting of two cross coupled CMOS inverters (M1 - M4) connected as a latch. The two inverters are highlighted in by the red and green triangle. The two MOSFETs (M5-6) allow the read and write access. The word line (WL) is controlling the current operation, and from the two bit lines (BL), the stored value can be read or written.

et al. [19] made deep analysis of the read weakness of different SRAM cells. During a read operation, it is possible that a stored "0" value can be overwritten by a "1" because of the mechanism of positive feedback. The measure of the stability of an SRAM cell to hold it's data against noise is called *Static Noise Margin* (SNM). It defines the noise voltage, which is required to flip the state of a cell [70, 19]. If this voltage level becomes too small, for example, because of a lower supply voltage, the cell may flip its inner state during a read.

The focus in this thesis lies on the six transistor cell because it is the most common. The circuit of the 6T SRAM cell is shown in figure 2.4. A 6T SRAM cell has three operation states, hold, write, and read. These states are controlled by the word line (WL). If the word line is "0" the cell value is held. If WL is "1" the cell is open for read or write operations. On a write, the value on the bit line (BL) overwrites the value in the latch and on a read operation, the value is read by sense amplifiers, which is comparing the value of the bit line and the negated bit line [49].

2.5 Instruction Decoding

The approach of decoding instructions is starting with an instruction fetch from the L1I cache. Depending on the opcode, it is forwarded to the right execution unit and later executed. In modern CPUs, this is far more complex, as can be seen by the Intel microarchitecture design shown in Figure 2.1. The process of decoding is called pipelining, and the single steps of a pipeline are done in parallel to increase the throughput [55].

2.5.1 Instruction Pipelining

Pipelining is one of the key features to allow fast computing. The idea is to split the execution of instructions into multiple steps and execute these steps in parallel to allow higher throughput. These steps are called stages in the pipeline [55]. Pipelining increases the processor's instruction throughput, the number of instructions executed per unit of time, but it does not reduce the execution time of a single instruction. Because of the introduced parallelism, the overall execution is faster.

The flow of an instruction is starting with the instruction fetch. The fetch unit is already quite complex since it decides which instruction to load next. If a conditional instruction is issued, the next instruction might not be known. In this case, the next program counter value has to be predicted. This is done by the branch prediction unit (BPU) [53]. A scheme of a fetch unit is shown in Figure 2.5.

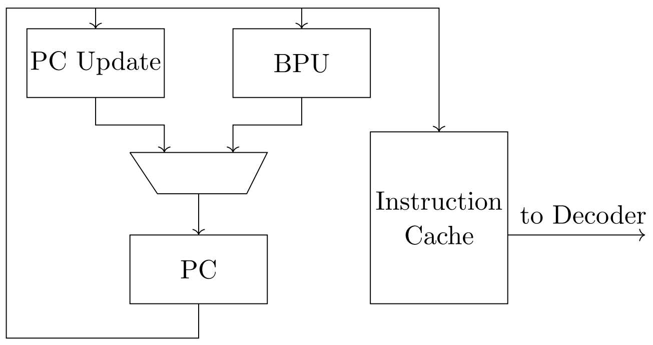

Figure 2.5: The program counter is either updated by the PC update unit or by the branch prediction unit. The PC value is used to fetch the next cache line from the instruction cache.

To keep a multistage pipeline busy, the BPU predicts the next cache line based on the previous branching history. Multiple algorithms have been invented to find a fast and reliable branch predictor [55]. Rolling back a wrongly predicted path is very expensive in terms of energy and time [53]. After the selection and decoding of the next instruction, the code is sent to the allocation queue and is forwarded in-order to the reorder buffer [31]. The IDQ is the end of the front-end and also the end of the in-order part. In the reorder buffer, register renaming, retirement and a number of optimizations are done like move elimination, ones idioms and zeroing idioms. These optimizations execute simple *μops* which for example just clear a register. To speed up the performance these *μops* are not sent to an execution unit [75].

The execution of instructions in parallel introduces new complexity and new hazards that have to be considered and resolved before the final execution [55]. The following enumeration lists dependencies of two instructions A and B, where A is in a sequential earlier order than B.

- data hazard occur when A produces a result needed by B
- structural hazard occur if A and B are using the same data location, for example A is writing and B is reading
- control hazard occur if between A and B is an extra branch instruction and it is not sure that B is executed after A

Hazards need to be resolved to avoid wrong executions. This is called a pipeline stall or a bubble. The pipeline has to be full all the time therefore bubbles are filled with "nop" instructions until the hazard is resolved [55]. Instructions issued before the stall have to be executed to resolve the stall, instructions issued after the stall are stalled as well, and during the time of resolve, no new instructions are fetched [55]. Stalls should be kept low to guarantee good performance.

Data hazards can be resolved by bubbles or operand forwarding. Forwarding is an optimization technique that delivers instruction A's result to the following instruction B by using a bypass. B does not have to wait for A to retire the result and can straight continue executing [60]. Structural hazards can be resolved by using a separate cache for instructions and data [55]. To avoid control hazards several techniques exist. On if-else conditions, it might be possible to reshuffle the compare instruction to get executed earlier in order to know the already taken branch. For loops, the branches can be avoided at all by loop unrolling. Furthermore, a branch predictor can be used to predict the outcome if a branch is taken or not. This is a hard task since indirect branches can have more than two possible targets [54, 58].

2.6 Dynamic Voltage and Frequency Scaling

Dynamic Voltage and Frequency Scaling is a technique that can reduce and increase the currently supplied power depending on the current workload. The basic elements of a CPU, CMOS transistors, allow a variadic operating range, which can be used to save the overall power consumption. If more work has to be done, the frequency and voltage can be increased to achieve a higher workload. Voltage and frequency levels are directly influencing the power consumption of a CPU, as can be seen in Equation 2.1.

$$P \propto C \cdot V^2 \cdot f \tag{2.1}$$

P is the power consumption, C is the capacitive load which is a function of the number of transistors and the technology used. f is the frequency, and V describes the voltage [55].

Because of the more comprehensive operating range of CMOS logic, CPUs can work in a more extensive range as well. Changing the supply voltage of a CMOS gate leads to an almost linear change in its operation delay. The energy a CMOS gate needs for operating is proportional to the square of the operating voltage [76]. This explains the quadratic effect of the voltage in Equation 2.1.

Frequency and voltage are also directly proportional to each other. A low voltage, for example, requires a low frequency in order to deal with the increasing CMOS delay. A low frequency, on the other hand, does not need a high voltage, throttling it results in a more significant power consumption saving. The DVFS algorithm has to assure that the combinatorial delay is never stretched to a critical extensive.

2.6.1 Synchronous Design

A synchronous design is the fundamental scheme of every modern CPU. Synchronous means that all state transitions occur periodically at the active clock edge. The internal circuit states or data are stored in flip-flops, also called registers, depending on the input they only change their output on a clock edge. In one clock domain, all clock signals have a fixed frequency and phase relationship. Figure 2.6 shows a simple synchronous design of two D-flip-flops, which are clocked by the same clock.

Two registers are called sequentially adjacent if they are connected with combinatorial logic, and no additional registers are between them [14]. The design of a synchronous circuit has to follow strict timing rules. Therefore it has to be done very carefully. *Clock skew* is defined as the timing difference between the arrival of the clock signal at two sequentially adjacent registers. The delay of the arrival time comes from the long way the signal has to pass through the electrical conductors. This can be seen in Figure 2.6

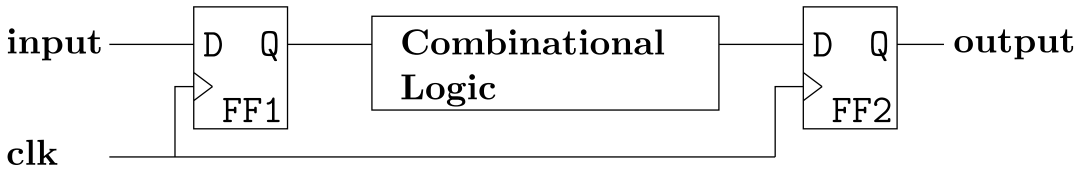

Figure 2.6: Synchronous design scheme of two sequentially adjacent D-flip-flops. The registers are storing the output of the combinatorial logic. This can be seen as the state of the circuit.

as the clock has to pass a longer way until it reaches FF2 [14].

Next to clock skew, the design also has to take into account the time a signal needs to propagate through the combinatorial logic. The clock period cannot be made faster than the signal, which has to pass through the logic. A logic circuit may consist of several logic gates like AND, NOR or OR. This execution delay t_c is coming from the time a transistor needs to change the inner state. Registers also have a timing constraint. During reading the input, it has to be constant for a specific amount of time called t_{setup}. The construction of the output needs time, as well. Before a combinatorial logic can use the output, an extra delay called t_{hold} needs to be added.

To avoid timing problems like glitches and to generate a stable circuit, these timing hazards have to be fulfilled. This gives every circuit a maximum clock frequency. Running faster can produce unpredictable behavior. The timings are visualized in Figure 2.7, showing that the maximum clock frequency is limited by the used components.

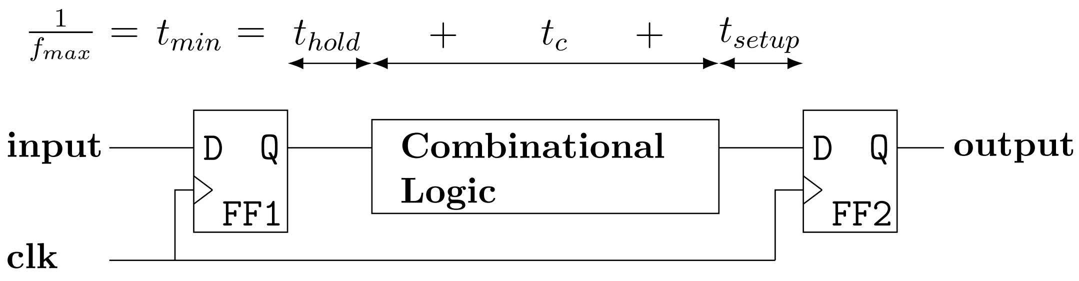

Figure 2.7: Timing constraints for sequentially adjacent registers.

The maximum clock frequency f_{max} is limited by the sum of all delays. The timing diagram in Figure 2.8 shows the response of the two flip-flop circuit from previous figures on a simple input. All timing hazards have been considered, and no undefined behavior occurs. As the combinatorial logic is not known, the timing behavior is unknown as well. This is marked with the switching action after a positive and negative edge.

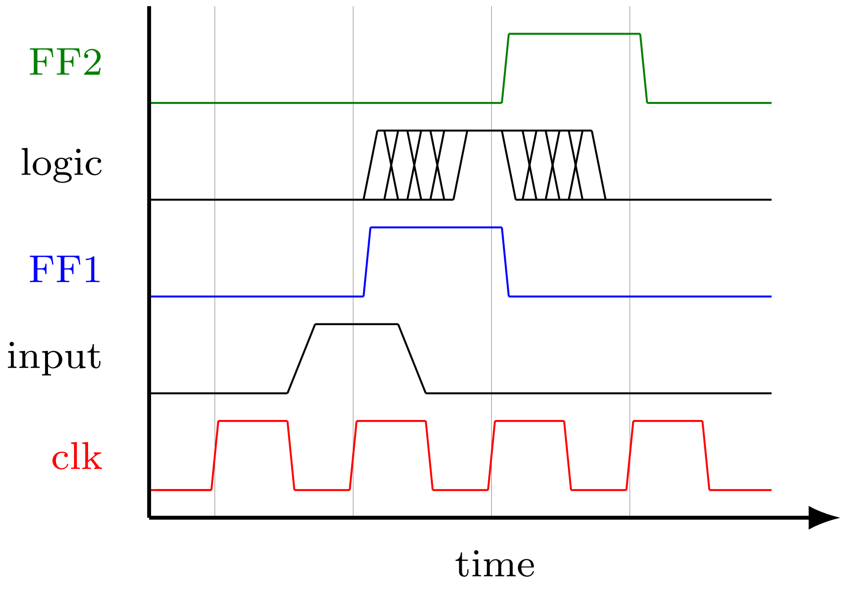

Figure 2.8: Timing diagram of a two register circuit. Using a faster clock may result in unpredictable behavior because of the constant execution delay from the logic circuit.

2.6.2 Fully Integrated Voltage Regulator (FIVR)

Before fully integrated voltage regulators, Intel used *package control units* (PCU) which were integrated on the die and distributing the power to the *power management architecture* (PMA) which is located on every core. PMAs are constantly collecting power consumption and temperature information and perform control actions like changing P-state and C-state transitions. Furthermore, it is forwarding the information to the PCU, which is keeping the external voltage regulator updated to adjust the power management [63].

Since the 4th generation of Intel CPUs, the Core is powered by fully integrated voltage regulators. FIVRs are highly configurable, allowing them to power a wide range of products from 3W tablets to 300W servers. FIVRs are also improving the battery live of mobile products up to 50% due to the better scalability. Furthermore, FIVRs are saving 75% of CPU die size and provide faster voltage transition times due to the embedded location. The voltage transition times are improved from the range of 100 microseconds for traditional voltage regulators to about 100 nanoseconds for FIVR technology [8].

The move from power control units to integrated voltage regulators is visualized in Figure 2.9. The integration of the regulators allows a more flexible regulator for the different cores on the CPU die instead of the same voltage for each cores [29, 33].

On the Skylake Generation, FIVR were removed and moved back to the motherboard because it was the bottleneck for overclocking in terms of extra heat generation. Over-

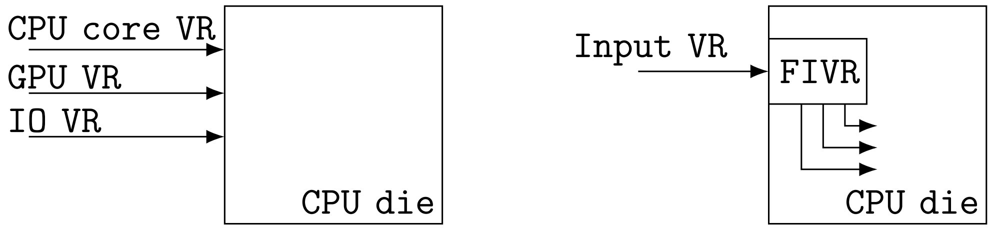

Figure 2.9: The switch from external voltage regulators, to internal regulators allows a faster and more optimized voltage supply for the different power planes in the cores.

clocking was considered to be more important than a flexible voltage adoption [75]. It is not sure if newer generations keep using PCUs or if Intel is trying the FIVR approach again.

2.6.3 Power Management

The power management logic of Intel CPUs is represented by the package control unit. The PCU builds the interface to higher power management hierarchies like the OS or BIOS. The current performance is handled in P-states. These are operating points of specific frequency and voltage levels. As explained in Section 2.6, frequency and voltage have a direct influence on the power consumption. Depending on the CPU workload, a P-state is chosen, which decreases the power consumption and keeps the performance unchanged [30]. Depending on the architecture, it consists of multiple P-states, P1 is the highest operating point where it guarantees a stable long term run, Pn is the lowest operating point, which is consuming the least energy. Turbo modes have their own P-state called P0. A state lower than Pn is used for critical conditions like thermal overheating [73, 32].

Since generation 6, Intel upgraded the P-state technology to Intel *Speed Shift Technology* (SST). Exposing the entire frequency range allows a smaller split between every operating point and gives more flexibility to the handling operating system [73].

P-states are operating states. If a CPU is running in idle mode, there exist C-states that describe how the CPU is handling an inactive state. C-states are operating per core. This allows the shutdown of a single core if it is currently unused [28].

2.6.4 Undervolting

Since the introduction of FIVR and also in later generations, which are not equipped with FIVRs, the adoption of the current operating voltage by the use of Model Specific Registers (MSR) is possible. Unfortunately, there is no official documentation to this

MSR interface, but we can find some Reverse Engineering work online. The documentation and tool is called ThrottleStop [21] which is a dynamic voltage frequency scaling tool for all Windows Platforms.

To change the voltage level of different power planes, it is necessary to write the changes to MSR register 336(0x150). In the Intel 64 and IA-32 Architectures Software Developers Manual is no entry for this register. Reading the Reverse Engineered resources and perform tests based on it, we can conclude to the following scheme of how the voltage is applied.

Each CPU has 5 power planes which can be set individually [16].

0: CPU Core

1: Intel GPU

2: CPU Cache

3: System Agent

4: Analog I/O

5: Digital I/O

For our cases, especially interesting are the planes 0 and 2. According to the Github repository linux-intel-undervolt [16], the CPU and Cache are sharing the voltage plane and cannot be set to different voltage values. The higher value is applied to both. We also verified this with our test CPUs. By only dropping the voltage of the cache, there was no measurable difference. Only a voltage drop of core and cache caused noticeable changes to the system.

The values written to the MSR register can be split into 5 parts as shown in Table 2.2.

Table 2.2: Description of MSR 0x150 value based on reverse engineering and analysis work.

Constant	**Plane index**	**Constant**	**R/W**	**Offset**
80000	0	1	1	1F000000

The constants are not known, but they have to be provided, the plane index describes which power plane is set, and the offset describes which voltage level is applied. The voltage level needs to be transformed to the correct offset before it can be applied. These steps are shown in Listing 2.1.

```c
uint64_t pack_offset(int32_t offset) {
  int64_t tmp = offset*1.024;
  tmp = (tmp & 0xFFF) << 21;
  tmp = 0xFFE00000 & tmp;
  return tmp;
}
```

Listing 2.1: Tranform a signed voltage value to an offset which can be directly written to the MSR register.

2.6.5 Frequency Scaling

With Intel Enhanced SpeedStep Technology, each processor comes with a P-state table containing frequency and voltage pairs. These tuples are describing fixed operating points, and they ensure that frequencies do not exceed the lower or upper bound [74]. The lower bound is called *Low Frequency Mode* (LFM) and the upper bound *High Frequency Mode* (HFM). To save power, the processor drops into a lower P-state.

In the Linux operating system, the cpufreq driver handles the frequency scaling. Compared to the intel_pstate driver, it allows more control about the frequency value. The decision about which frequency is applied is handled by the currently chosen governor. There are six governors in the Linux kernel, performance, powersave, userspace, ondemand, conservative, and schedutil. Performance and powersave are running continuously on frequency limits independent of the workload. The userspace governor allows the user to specify the current frequency. Ondemand, conservative and schedutil are using different techniques to set the current frequency depending on the workload [45].

2.7 Intel Software Guard Extension

Intel's Software Guard Extension (SGX) is an instruction set architecture extension with the goal of providing integrity and confidentiality to security-sensitive operations on computers where privileged software is potentially malicious [13]. Performing secure remote operations on an untrusted remote computer owned by a third party, for example, on Cloud environments, is an unsolved problem in secure remote computing. Cryptographic algorithms tackle this problem only partly since the running code can be extracted or memory can be read and modified from higher privileged processes running on the same computer. The strongly reduced attack surface of an SGX application does not have these drawbacks and provides more security. The SGX feature is not enabled by default on a supportive CPU it has to be enabled in the basic input/output system (BIOS). [13].

A processor with enabled SGX support protects the computation of a program by storing it in an enclave (secure container). The enclaves code and data are isolated from the outside. Other programs and also the operating system and the hypervisor cannot access the isolated memory. SGX acquires Processor Reserved Memory (PRM), which stores the enclave data and is protected by the CPU against non-enclave accesses [37].

An SGX application live cycle is depicted in Figure 2.10. A reasonable application for using the trusted execution extension might be hosting cryptographic algorithms. Storing keys and the encryption code in SGX deals with the problems of storing plain text keys in the source code and being vulnerable to code injection and other attacks.

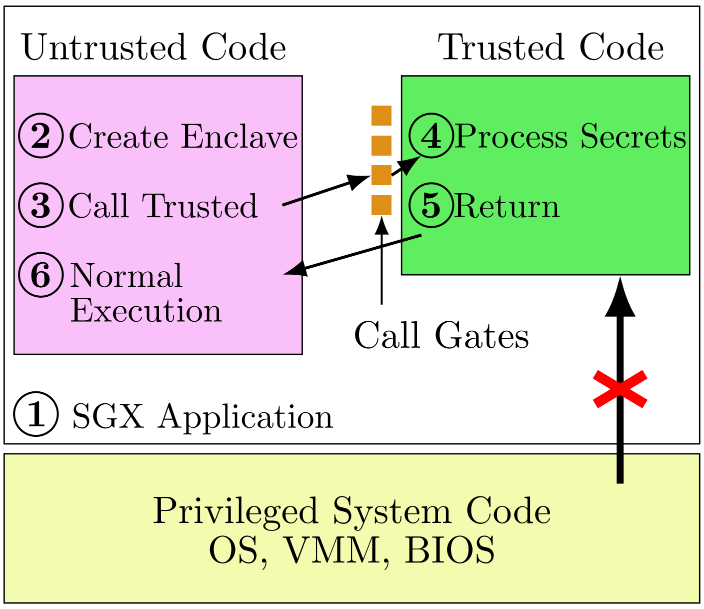

Figure 2.10: Example call from untrusted code to trusted code via the SGX interface.

A program is built with a trusted and untrusted part (1), at startup or runtime the application is creating the enclave (2) which is placed in a trusted memory region. A trusted function is called (3), and the execution is moved to the enclave. Inside the enclave (4), all process data is visible in clear. When the enclave has finished its operations, the execution is passed back (5) to the untrusted part of the application. After the enclave call (6) normal execution continues.

To better understand the protection mechanism of SGX, a deeper dive into the technology is necessary. The memory an enclave uses is structured in pages of 4kB sizes. The CPU tracks each page state in the Enclave Page Cache Map (EPCM) and ensures that each page belongs to exactly one enclave. This split of the SGX memory allows running multiple enclaves on the same system. Each processor is expected to handle between 5 to 20 enclaves at once since there is a hard size limit of space for protected memory which is either 64 MB, or 128 MB. The memory page structure is called Enclave Page Cache (EPC). Figure 2.11 shows the memory design of SGX.

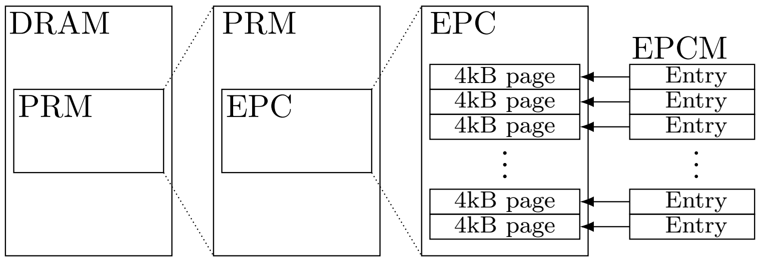

Figure 2.11: The PRM memory is stored protected in the DRAM and cannot be modified from outside. All data from different enclaves is inside the PRM.

It is worth mentioning that on an initial enclave load from untrusted system software, the CPU is asked to copy the unprotected data on a secure page inside the PRM. Out of this, it follows that the untrusted software knows the initial enclave state [13].

The security of an enclave is achieved by the encryption and decryption of the data and code. When data is not used, it is stored in the main memory. Only when accessed, the data is decrypted by the memory encryption engine (MEE). The MEE is the main hardware modification of the new technology. Intel [34] does not document the internal functioning. Previous work [13] shows that it relies on Merkle trees to provide confidentiality. The MEE is located in the processor's memory controller. This means that in the memory hierarchy, it is below the caches. It follows that enclave memory is stored unencrypted in all CPU caches. The key for memory encryption is chosen at random and also randomly changes on every power cycle like reboot or resuming from hibernate mode. Furthermore, the random key is stored in the CPU and is not accessible from any part [13].

Moving the execution flow from regular userspace code to an enclave cannot be done by a simple call or jump instruction, newly added instructions are used to continue the execution in the enclave. An entry call in an enclave performs several protection checks before the execution is forwarded to the enclave. These access checks are crucial for the security guarantees of SGX, especially since the data in the caches is unencrypted.

The new trusted environment from Intel also has some drawbacks. Since it is not possible for non-enclave code to access enclave memory, although, it is possible for enclave code to access non-enclave memory, this can lead to new types of malware [66]. Another future problem might arise from the unencrypted data in the caches. If software attacks exploit side-channel techniques on the cache, this breaks all security SGX provides.

2.8 Asymmetric Cryptography

Asymmetric cryptography or public-key cryptography uses a scheme of two keys, a public key that is widely known and a private key that is only known to the owner.

In comparison to symmetric cryptography [39], the encryption and decryption keys are different. The generation of such keys has to assure that it is a one-way function based on cryptographic algorithms [64, 17]. Messages are encrypted using the public key and can be decrypted by the private key to assure only the owner of the private key can read the message.

Asymmetric cryptography is also used as a signature scheme by decrypting a hash of a message with the private key. Sending the decrypted hash along with the message allows the receiver to verify the message by encrypting the signature with the public key and comparison of the hash.

2.8.1 RSA

RSA [62] is a widely used asymmetric cryptography scheme and was one of the first developed. The one-way function in RSA is based on the difficulty of factorizing the product of two big prime numbers [52]. The advantage of RSA is to encrypt a message without exchanging a key. A disadvantage of RSA, however, is that the algorithm is rather slow due to heavy multiplications. The original algorithm is described in the following enumeration [47].

1. $n = p \cdot q$ where p and q are two large prime numbers and the bit size of n defines the security level of the algorithm
2. compute $\phi = (p - 1) \cdot (q - 1)$
3. choose an integer e (public exponent) such that, $1 < e < \phi$ and $gcd(e, \phi) = 1$
4. compute d (secret exponent) such that, $1 < d < \phi$ and $e \cdot d \equiv 1 \bmod \phi$

The public key is now the pair (n,e) and private key which has to be kept secret is (d,p,q). For encryption of an arbitrary message the public key is needed as shown in equation 2.2.

$$c = m^e \; mod \; n \tag{2.2}$$

The message m must be converted in an integer smaller than the modulus n. The resulting ciphertext is noted as c. The decryption of the cipher text is shown in equation 2.3.

$$m = c^d \; mod \; n \tag{2.3}$$

The difficulty in breaking this scheme stems from factorizing n in p and q, for example, if this is achieved, an attacker could calculate phi and also the private exponent d.

Because of the high computational effort of the exponentiation, RSA is not used for encrypting large data blocks. The preferred usage is for the exchange of a symmetric key, which is later used for the encryption of the data [52].

The RSA algorithm is also used for signature schemes. Encrypting a hash of a message using the private key generates a signature that can be verified by the public key. Every receiver of the signed message can assure it was sent by the expected sender and is not modified.

2.8.2 RSA-CRT

An improved scheme for RSA is using the Chinese Remainder Theorem(CRT) [46] which results in a four times faster computation. This is highly used in embedded systems where computing power is limited. Considering the RSA decryption "$m = c^d \, mod \, n$" with the private key (d, n). The private key is not as convenient as the public key. For the public key, a value with a low number of one bit can be chosen to gain fast encryption. The private key is at a length of the modulus n, and approximately half the bits are ones. This is a high computational effort proportional to k^3 (k the number of ones) for the decryption [48].

The RSA-CRT algorithm addresses this problem of computational effort by pre-processing the private key to gain faster encryption. The steps of pre-computation, encryption, and decryption are shown in the paragraphs below.

RSA-CRT pre-computation

The steps of pre-computation are shown in the equations 2.4, 2.6 and 2.7, where p and q are the secret prime numbers, e is the public exponent and d denotes the private exponent.

$$d = (1/e) \, mod \, (p-1) \cdot (q-1) \tag{2.4}$$

$$dP = d \, mod \, (p-1) \tag{2.5}$$

$$dQ = d \, mod \, (q-1) \tag{2.6}$$

$$qInv = (1/q) \, mod \, p \tag{2.7}$$

RSA-CRT encryption

The encryption uses the results from the pre-computation steps and is applying the two pre-computed primes one after the other on the message m. All steps are shown in the equations 2.8 to 2.11.

$$c_p = m^{dP} \, mod \, p \tag{2.8}$$

$$c_q = m^{dQ} \, mod \, q \tag{2.9}$$

$$h = qInv \cdot (c_p - c_q) \, mod \, p \tag{2.10}$$

$$c = c_q \; + \; h \cdot q \tag{2.11}$$

RSA-CRT decryption

Using the ciphertext c, the original message m can be obtained by using the regular RSA equation shown in 2.12.

$$m = c^e \, mod \, n \tag{2.12}$$

The private key (p, q, dP, dQ, $qInv$) can be stored and reused for later operations, resulting in a overall faster encryption or signature generation.

Fault injection Vulnerability

The RSA-CRT variant is known to be very prone to fault attacks during the computation of a signature or ciphertext [5]. A successful injection can lead to a leak of one of the private primes. The problem in the algorithm lies in the sequentially applied modular exponentiation of the primes, as shown in the equations 2.8 and 2.9. If in one of the two computations a fault appears the other private prime can be calculated by applying the greatest common divider as it is shown in Chapter 5. Research [72] has been done to overcome this weakness.

2.8.3 Sliding Window Exponentiation

Modular exponentiation is the critical computation path in asymmetric cryptography in terms of timing. To improve this bottleneck, different exponentiation algorithm exists. The focus in this thesis lies on the Sliding Window Exponentiation technique since it is used in the open-source cryptographic library mbed TLS [3]. This library is used in later sections for security analysis.

Sliding Window Exponentiation is using the fact that the exponent can be split to reduce the number of multiplications needed [78]. For example, an exponent represented in binary format can be split, as shown in equation 2.13.

$$base^{1011\ 0010_{(2)}} = base^{1011_{(2)} \cdot 2^4} \cdot base^{0010_{(2)}} \tag{2.13}$$

This split of the exponent reduces the amount of multiplications due to pre-computation of the values $base^2, base^4, base^8, base^{16}, \cdots$. The value $base^{16}$ in 2.13 is already computed and does not have to be calculated during the operation. This method has an extra pre-computational overhead, but the overall execution has fewer multiplications.

Instead of a fixed size split as used in m-ary methods, the sliding window exponentiation allows a variable size, called the window. The size of the window is adjustable in the hope of finding windows that contain only zero bits and result in a computation of pre-calculated values [7].

Mbed TLS is using the sliding window algorithm 14.85 from the Handbook of Applied Cryptography [50], which is searching for zero bits in the exponent and whenever found it is using the pre-computed values. Otherwise, it searches for the longest bitstring, which still fits into the predefined window size and ends with a high bit. The algorithm 14.85 is shown in Algorithm 1.

```
INPUT: $g, e = (e_t e_{t-1} \cdots e_1 e_0)_2$ with $e_t = 1$, and an integer $k \geq 1$.
OUTPUT: $g^e$
    1. Precomputation
       1.1 $g_1 \leftarrow g$, $g_2 \leftarrow g^2$
       1.2 For $i$ from 1 to $(2^{k-1} - 1)$ do: $g_{2i+1} \leftarrow g_{2i-1} \cdot g_2$
    2. $A \leftarrow 1$, $i \leftarrow t$
    3. While $i \geq 0$ do the following:
       3.1 If $e_i = 0$ then do: $A \leftarrow A^2$, $i \leftarrow i - 1$
       3.2 Otherwise ($e_i \neq 0$), find the longest bitstring $e_i e_{i-1} \cdots e_l$ such
           that $i - l + 1 \leq k$ and $e_l = 1$, and do the following:
              $A \leftarrow A^{2^{i-l+1}} \cdot g(e_i e_{i-1} \cdots e_l)_2$, $i \leftarrow l - 1$
    4. Return($A$)
```

Algorithm 1: Exponentiation algorithm which uses the sliding window technique.

Chapter 3

Attack Overview

In this chapter, we describe our idea of inducing bitflips in the instruction cache. Our approach is to stretch the combinatorial delay until a correct computation cannot be assured anymore. This can be done by setting the frequency of the processor to the maximum and decreasing the supply voltage at the same time. This approach is increasing the operation time of the CMOS gates, but the high frequency expects fast transition times, which is a contradiction and might lead to faults in the cache.

We target the instruction cache by continuously fetching instructions. A fault might lead to a modified instruction. This modification is detected by the parity bits check in the cache and invalidates the result and query the level 2 cache for the instruction. These extra operations can be measured by using fast counters and result in an overall slower execution.

3.1 Attack Model

To undervolt a system, root privileges are required. This limits the attack surface in a real-world scenario. Having already root privileges, it does not make sense to target other software components or the operating system. A valid attack scenario would be to target Intel SGX. SGX states, to stay secure even if the hosting operating system is considered malicious. Producing a fault in an instruction which is leading to a wrong computation or an exception, is violating the SGX statement. Modifications from outside of the enclave should not be possible. However, modifications on the frequency and the supply voltage also affect the code running inside an enclave. This gives us two knobs that we can control to influence the execution of the enclave.

In our attack model, the victim computer needs to fulfill certain assumptions. It has to be an Intel CPU not older than generation 6. Access to the model specific register 0x150 must be allowed, and SGX has to be enabled. Furthermore, to successfully deploy our attack, an SGX application has to run on the victim. Lastly, we also assume that we have root access. It is important to state that we do not need physical access. This attack can be done remotely as well.

From the attacker perspective, an accurate counter has to be established on the victim machine. The SGX application needs to be queried frequently, and the response time and results have to be checked. An increase in the response time or a different response assumes a successful attack.

All our test cases are executed on a Linux (Ubuntu 18.04) operating system. The test cases are working the same on a Windows operating system. Only the frequency needs to be modified in a different way, as explained in later sections. Since the cpufreq interface is called differently on Windows bases operating systems.

In the first test round, no test cases contain enclave code to allow building an easier test framework. If bitflips are induced with non-enclave code, they are induced by enclave code as well, since the code is executed on the same CPU and stored in the same cache.

3.2 Stretching Combinatorial Delay

As described in Section 2.6.1, increasing the frequency to an unexpectedly high value for the system can lead to an unpredictable behavior because of a constant combinatorial delay. Furthermore, from Section 2.6, we know that CMOS technology needs more time to change the state when the supply voltage is lowered. These two factors are leading to the conclusion that a high frequency, which requires fast transition times and a low voltage, which requires a low frequency in order to have enough time for a change, might not be compatible with each other. This may induce faults to operations in the instruction cache.

To force a system to use a constant frequency the Intel P-states have to be disabled. This can be achieved by disabling the P-states in the boot configuration and using the frequency governor userspace. This governor allows setting the frequency to a constant value in the supported range. The possible frequency range is documented in the CPUs datasheet, but can also be tested by running a core at a high workload and setting the frequency to different levels and measure which are applied. Using the userspace governor on our test CPUs, we measured the operating points shown in Table 3.1.

Table 3.1: Operating points on a linux host system using the userspace frequency governor. The first row shows the LFM frequencies and the last row is the HFM. This table also shows the dependency between the frequency and the voltage.

i3-7100U		**i5-6200U**	
Frequency [GHz]	**VCore [mV]**	**Frequency [GHz]**	**VCore [mV]**
1.59	0.698	0.89	0.675
1.69	0.718	1.09	0.704
1.89	0.750	1.29	0.732
1.99	0.769	1.39	0.741
2.09	0.788	1.49	0.755
2.29	0.826	1.69	0.789
2.39	0.845	1.79	0.804
		1.99	0.840
		2.09	0.859
		2.19	0.874
		2.69	0.972
		2.79	0.992

Table 3.1 shows a clear dependence between voltage and frequency. By applying undervolting to these operating points we might stretch the combinatorial delay until a critical operating point is reached. In Section 4, we perform several high workload tests on our two test CPUs. These test cases keep the CPU busy while a second program is constantly undervolting the core until the machine freezes or becomes unstable. This gave us good stretched operating points, which still allow correct code execution, but a slightly lower voltage might already freeze the whole system. We use these critical points in later chapters to run further tests on the instruction cache.

3.3 Cache Hammering

As explained in Section 2.4, the SRAM cell has a read weakness that is the target we are trying to address. Aiming only for the read operation is good enough for the instruction cache, we can force a high usage by a high number of instruction fetches. To achieve a continuous fetch of instructions from the instruction cache, we have to avoid the CPU optimization of fetching already decoded instructions from the μop-Cache.

In Chapter 4, we research the behavior of the μop-Cache and analyze when and which instructions are loaded from it and which instructions are never stored in it. A simple approach to avoid the μop-Cache would be to increase the size of a loop until it is too big to be stored in it.

To determine into how many μops a code block gets decoded, the Intel Architecture Code Analyzer (IACA) [35] is used. IACA statically analyzes code snippets for data dependency, throughput, and latency. We only use the feature for converting our test cases into μops to check if it is possible to be stored in the μop-Cache or the LSD.

```
  xor rax, rax
  mov rcx, 0x1234

  ; IACA start marker
  mov ebx, 0x6f
  db 0x64
  dw 0x9067

loopstart:

  ; do some operations

  inc rax
  cmp rax, rcx
  je exit
  jmp loopstart

  ; IACA end marker
  mov ebx, 0xde
  db 0x64
  dw 0x9067

exit:
  mov rax, 0x3C      ; exit
  syscall
```

Listing 3.1: Extracted IACA start and end markers in a nasm assembly example code. Noting that the marker is changing the ebx register, this needs to be considered if the marker is left in the code during execution.

IACA is only analyzing the code between the start and end markers. These have to be put around the test loop in the test cases to get the number of *μops*. This gives us a second verification possibility next to performance counters to verify the plausibility of our results.

Listing 3.1 shows a sample test case, including the IACA marks around the loop.

3.4 Experimental setup

The victim machine is running in an unstable operating point during our attacks. The experimental setup has to ensure that the victim does not crash during testing and also that the obtained results are reliable and reproducible. The setup needs to deal with interference from the run-time environment, hardware, and software.

To achieve this, we run the test program on an isolated physical core. The attacker, in the test case scenario, the undervolting program, is running on a different physical core. The frequency is constant for all test cases if not mentioned differently. Whenever timing differences are measured, the scheduling behavior of the Linux operating system is considered by giving higher priorities and performing a yield before each measurement. To measure the execution time, we implemented our own counter. This counter runs on the same physical core as the undervolting program but on a different logical core to get high accuracy. This counter contains a simple loop that increments a global 64 bit variable. With the start and end difference of this variable, we can compare different execution times.

A successfully induced flip can be noticed by a slower execution time, measured by the self implemented timer as well as by a different operation result. All test cases perform a certain operation using the same instruction, which should lead to the same result on every run. For example, if one instruction is fetched with a fault, another instruction is executed, and a different result is obtained.

3.5 Attack Aim

The aim of this thesis is to produce bitflips in the cache, due to the previous explained findings, we decided to focus our research on the instruction cache. This is our first target also because the instruction cache does not use error correcting codes (ECC) [55]. This is stated with the conclusion that the instruction cache is read-only and does not need error correction, although it is still equipped with parity bits where one parity bit is testing 6 data bits.

This means that a successfully induced bitflip should result in a measurable timing difference in the execution. A slower execution should occur because of a failing parity bit check and in the resulting fetch from the next level cache. The goal is to introduce wrong operation results by inducing two bitflips in one parity check area.

If a wrong computation result occurs during our tests on specific instructions, this shows that it is also possible for us to induce wrong computation results inside an SGX enclave. Faulting operations inside an enclave can have a critical effect on the security of the system when the enclave is running cryptographic applications, for example, a signature generation application. A wrong intermediate arithmetic operation can lead to a completely different signature. Obtaining various signatures for the same message can lead to an extraction of the private key [1].

Chapter 4

Analysis Results

In this chapter, we present our results and findings, which we discovered during the detailed analysis and search for bit flips. All tests were performed on an Intel CPU i3-7100U and i5-6200U. Section 4.1 contains the results of testing when a decoded instruction is stored in the *μop*-Cache. Section 4.2 states the stability analysis of how the test CPUs are behaving under certain undervolting levels. In the last part, we combine the analysis results and start exhaustive tests on the instruction cache.

4.1 *μop*-Cache

In this section, we present our analysis results of the behavior of instructions depending on the *μop*-Cache. All tests were performed on our two test CPUs and on an isolated core and pinned to it for execution. To shorten the result tables, we present only results from the i3-7100U. The Kabylake architecture is assumed to be very similar to the Skylake architecture. Almost all observed results were similar, different behavior of test cases is mentioned.

4.1.1 Instruction Search

As stated in Chapter 2.3, decoded instructions are stored in the *μop*-Cache if certain conditions are fulfilled. Because of our approach to hammer the instruction cache, we have to fetch as many instructions as possible from it instead of fetching from the *μop*-Cache. To achieve this, we came up with two ideas. We can increase the loop size until the continuous basic block is too big to be stored in the *μop*-Cache, or we use instructions that get decoded into more than 6 *μops*.

Increased loop size Increasing the loop size of the test programs avoids the storage in the *μop*-Cache and result in a fetch from the instruction cache. From where the instruction is getting fetched can be measured by using performance counters. By the counters "*μops* issued from *μop*-Cache" and "*μops* issued from MITE" we are able to distinguish from where the instructions is getting fetched. To better verify the performance counters, we also track the L1I accesses, where we should see a notable rise once the loop size reached a certain level. Figure 4.1 is showing a test case with an increasing loop size.

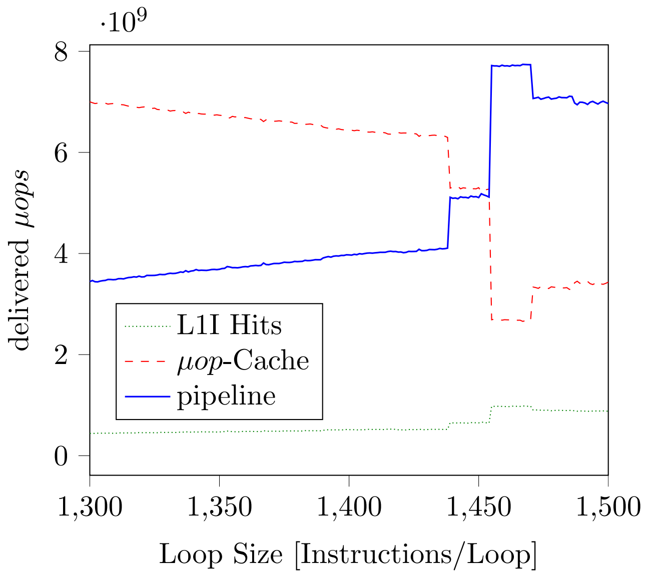

Figure 4.1: Continuously increasing the loop size leads to an instruction fetch from the legacy pipeline. This can be seen at approximately 1.500 increment instructions. At this loop size, the fetch from *μop*-Cache is switching to the pipeline.

After around 1.500 instructions, the *μop*-Cache is full and cannot store more decoded instructions. This fits with the defined *μop*-Cache size of 1.536 *μops*. The increment instruction is very simple and gets decoded into one *μop*, using more difficult instructions leads to more stored decoded instructions in the *μop*-Cache. This sums up that any continuous block exceeding the *μop*-Cache size leads to a fetch from the instruction cache.

Long Instructions The Intel IA-32 Architecture Optimization Reference Manual [31] states that Instructions which get decoded into more than 6 *μops* are not stored in the *μop*-Cache due to the architecture of the *μop*-Cache that each cache way can hold up to 6 *μops*. Using this knowledge, we focus on heavy instructions like division, which decodes to 8 *μops* and are therefore never stored in the *μop*-Cache. We verified this with the division instruction, which is shown in Figure 4.2.

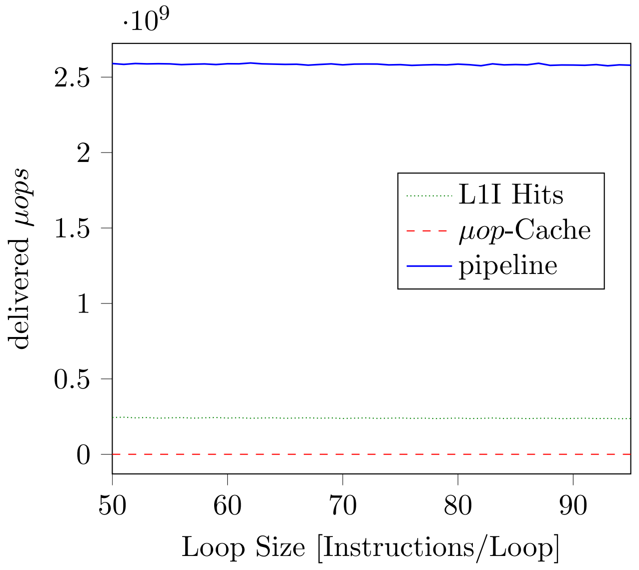

Figure 4.2: The test case is containing an increasing number of division instructions. Already at a low number of instructions, they are not issued from the μop-Cache.

As can be seen on the division instruction, which gets decoded into 8 μops, it is never fetched from the μop-Cache. Independent, if it is in a loop or not, the execution is always slower than an execution from the μop-Cache.

Unexpected behavior We also tested several other instructions and found that some are never stored in the μop-Cache even if they get decoded into less than 6 μops. This is a contradiction to the Intel IA-32 Architecture Optimization Reference Manual [31]. For example, as shown in the following Figure 4.3, we see that an increment on a memory location is never issued from the μop-Cache.

This is an interesting and not expected behavior of this instruction. We see that already at a small loop size with 100 instructions, which get decoded to 400 μops, not getting fetched from the μop-Cache, which should happen as explained in Section 2.3. Analyzing the code snippet using IACA (Intel Architecture Code Analyzer) shows that one increment is decoded to 4 μops. This result is shown in Table 4.1.

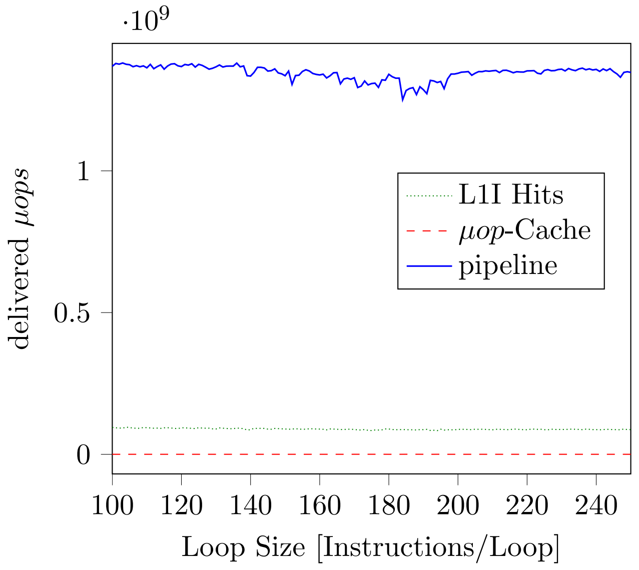

Figure 4.3: The analysis of a test case with an increasing number of increments on a memory location instruction shows that they are never stored in the *µop*-Cache.

Table 4.1: IACA analysis report of an increment on a memory location instruction. We can see a repeating behavior after three instructions, this is repeating until the end of the loop.

Num Of	**Ports pressure in cycles**								
Uops	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Instruction**
4^	0.3	0.3		1.0	1.0	0.3	0.2	1.0	inc dword ptr [rbp-0x14]
4^	0.2	0.3	1.0	1.0	1.0	0.3	0.3		inc dword ptr [rbp-0x14]
4^	0.3	0.2	1.0		1.0	0.3	0.3	1.0	inc dword ptr [rbp-0x14]
4^	0.3	0.3		1.0	1.0	0.2	0.3	1.0	inc dword ptr [rbp-0x14]
4^	0.3	0.3	1.0	1.0	1.0	0.3	0.2		inc dword ptr [rbp-0x14]
4^	0.2	0.3	1.0		1.0	0.3	0.3	1.0	inc dword ptr [rbp-0x14]

An instruction that gets decoded into 4 *µops* should be according to the Intel Optimzation Guide [31], stored in the *µop*-Cache. Due to this finding, the question arises if there exist more instructions that are never stored in the *µop*-Cache, resulting in slower execution. Our test setup to search for further interesting instructions consists of different arithmetic operations located in a loop. The increment instruction is a simple arithmetic operation, and therefore we consider other arithmetic instructions in the first test round. It might be that the behavior is typical for this type of instruction. The different loops contain 100 instructions of the same type to avoid storage in the LSD. To ensure that the instruction is fetched from the *µop*-Cache we use performance counters. The results of this analysis and the tested instructions are listed in Table 4.2. To keep the table easily readable, all numbers are divided by 10^4 and only integer numbers are shown. We are not interested in the exact numbers only in the difference from which path the *µops* are getting fetched.

Table 4.2: The analysis of different instructions in a basic block containing 100 of the same type shows some unexpected behavior. Independent from the number of *μops* it is getting decoded to it might be stored in the *μop*-Cache or not. Especially instructions which operate on a memory location seem to not get stored in the *μop*-Cache.

Instruction	*μops*	*μop*-**Cache**	**MITE**	*μops* **Executed**	**L1I Hits**
inc eax	1	520389	324	513511	62
inc [m]	4ˆ	4	154293	207615	10265
dec eax	1	517312	303	520532	65
dec [m]	4ˆ	4	155400	206496	10208
add eax, 0x1	1	509753	203	512940	73
add [m], 0x1	4ˆ	108240	262	214112	70
sub eax, 0x1	1	515521	214	518506	67
sub [m], 0x1	4ˆ	109547	174	216969	56
neg eax	1	517426	284	518637	72
neg [m]	4ˆ	3	108312	215885	10731
not eax	1	500476	323	507530	78
not [m]	4ˆ	3	151216	201256	10041
rol eax, 0x1	2	3	575146	585517	40043
rol [m], 0x1	5ˆ	3	220309	273985	10788
ror eax, 0x1	2	3	522301	529408	36410
ror [m], 0x1	5ˆ	3	219619	277026	10952
shl eax, 0x1	1	529521	233	534299	67
shl [m], 0x1	4ˆ	3	153410	206707	10247

Taking the results from Table 4.2, we can see that instructions operating on a memory location should be our preferred choice. The ”rol” and ”ror” instructions are performing best for our purposes, independent if they are accessing a memory location or not. They are always issued from the legacy pipeline and have a high hit rate on the instruction cache.

We compare this newly discovered instructions to our approach of increasing loop size and too big instructions. All test cases run for 1 second monitored by performance counters. Table 4.3 depicts the results of the comparison. As before, all performance counter values are divided by 10^4.

Table 4.3: A detailed performance comparison of the rotate left instruction, the increment instruction and the division.

Instruction	**Loop Size**	*μops*	*μop*-**Cache**	**MITE**	*μops* **Executed**	**L1I Hits**
inc eax	1600	1	99.980	139.584	240.028	26.315
rol eax, 0x1	100	2	533	446.152	460.517	44.543
div [rsp]	10	8	532	22.809	210.412	26.464

The comparison in Table 4.3 shows that the ”rol” instruction is accessing the instruction cache twice as often in the same amount of time as the other two instructions. Also twice as many *μops* are executed, leading to an overall higher workload for the CPU. The higher amount of executed *μops* may be explained because using only ”rol” instructions almost no *μops* are issued from the *μop*-Cache. Therefore no switches between the pipeline and the *μop*-Cache occur, leading to a faster continuous execution from the pipeline. In comparison to the division instruction, the difference in executed *μops* can be explained by the slow execution of a division due to its complexity, which is shown in the amount of *μops* it gets decoded to.

4.1.2 Instruction Behavior

During the analysis steps from the previous section, we made some more interesting findings according to the behavior of the rotate instruction. Our test cases showed that depending on the immediate value the instruction is fetched from the *μop*-Cache or from the instruction cache. We verified this on our two test CPUs. Both showed the same behavior.

We tested with the immediate values 1, 2, and choosing the cl register. The IACA tool gives two times the same result, both instructions are decoded into 2 *μops*. Although performing our test cases, we can not verify this result. The test case has the same structure, 100 instructions of the same type in a loop. Table 4.4 shows the results of the test runs.

Table 4.4: The loop in the test cases performed 10.000.000 iterations, each of it executing 64 rotate right instructions. Interestingly not all test cases have the expected amount of executed *μops*.

Instruction	**Bytes**	*μops*	*μops*-**executed**	**MITE**	*μop*-**Cache**
ror rax, 0x1	48D1C8	2	1.312.002.111	1.310.927.553	886.213
ror rax, 0x2	48C1C802	2	672.001.413	988.397	670.818.530
ror rax, cl	48D3C8	3	1.953.011.480	1.951.519.965	982.646

As the test cases are showing, changing the immediate value is also changing from where the instruction is loaded. We verified this unexpected behavior on our two test CPUs i5-6200U, i3-7100U. On both, a change to an immediate unequal to 1 lead to a fetch from the *μop*-Cache and only an immediate value of 1 is issued by the pipeline.

If we manually calculate the number of instructions that should get executed according to IACA, we get $10^7 * 2 * 64$, which leads to $1.28 * 10^9$ *μops*. This fits well with the immediate value 1, which has a few more executed *μops*. These extra instructions can be explained by the startup and tear down overhead of a program.

If we increase the immediate value, the results show that we get only half as many executed *μops*. This lets us assume that the instruction is decoded into one *μops* and not into two, as the IACA report showed. This is an unexpected result and cannot be fully explained.

By using the cl register instead of the immediate value, we obtain the expected number of executed *μops*. Table 4.4 shows a similar amount of overhead by using the cl register compared to an immediate value of 1.

4.2 Stability Analysis

In this section, we present our results on stability analysis on our test CPUs by different undervolting levels. We show with the following tests that we can reach and hold unstable operating points and use them to produce faulty operation results. The results in this section are mainly obtained from the i3-7100U if not stated otherwise.

4.2.1 Undervolting

As described in Section 2.6, for performance and energy saving purposes, the CPU allows the operating system to adopt the current voltage supply level and also the current frequency. These two knobs can be controlled by a userspace program and are the base for our following test cases.

Using MSR 0x150 for undervolting allows a program to reduce the voltage supply to a level which halts the CPU due to a too low supply voltage. In our first analysis steps, we focus on finding this critical undervolting level, on avoiding it in later test runs.

The first test case consists of a simple endless loop that was working as a heartbeat for the system to measure if it is alive or frozen. The voltage was decreased manually to give the system enough time to adapt and also to test the undervolting level by starting different applications, for example. The results of these tests can be seen in Table 4.5.

Table 4.5: Results of different undervolting levels, reducing the voltage too much leads to an instant halt of the system, while other operation levels may lead to a halt after some time of running stable.

Unvervolt Level	**Stability Result**
-150 mV	instant halt
-140 mV	halt after few seconds
-134 mV	crashes running applications, Window Manager, Visual Studio Code...
-132 mV	too unstable for proper testing, kills SSH-connections
-130 mV	no faulty behavior detected

4.2.2 Frequency Scaling and Undervolting

As stated in the background, a high frequency must have a high voltage in order to guarantee the correct execution of instructions. Our tests were focusing on creating a large gap between frequency and voltage by holding the frequency high and undervolt as much as possible.

During testing, we noticed that a low frequency combined with a low voltage does produce far more instability than a high frequency with a low voltage. This was tested by a signature generation test case, which is calculating RSA signatures in a loop until a wrong signature is calculated, which equals a fault in one of the computations. We can not explain why undervolting with a low frequency produces more faults than with a high frequency, according to our research in Chapter 2. The result of this stability test is shown in Figure 4.4.

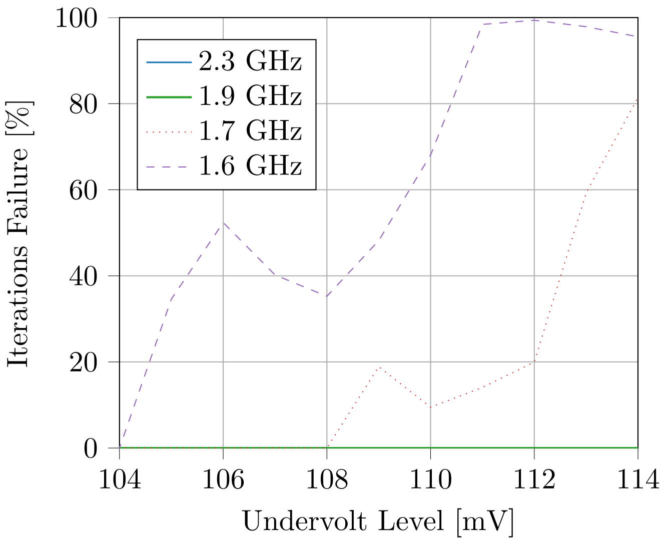

Figure 4.4: Behavior analysis of voltage and frequency scaling. It shows that the combination of a low frequency with a high undervolting level is leading to the most faults.

For all our following test cases, we used an operating point with a constant 1.6GHz frequency and an undervolting level between -108 and -110. Using higher undervolting levels frequently broke SSH-connections, so we only used higher levels in certain test scenarios.

A discovery that was obtained during testing is that setting the frequency to a lower level than 1.6GHz decreases the amount of wrongly calculated signatures. This can only be explained by the fact that the gates in the CPU have enough time for a state change.

4.3 Cache Hammer

The research from the previous sections gives a good starting point for the cache hammer approach. Attacking the instruction cache can now be done by running our chosen instructions in a loop and applying undervolting and frequency scaling during the execution. Introducing bitflips in the instruction cache should lead to a parity bit check fail and a noticeable execution delay. This delay should be measurable using performance counters. This approach builds our first test case scenario. Our second scenario is running the test case in an endless loop without considering performance counters. A successful induced flip might also lead to a wrongly fetched instruction and, therefore, in a wrong computation result.

```
  mov r11, 0x6665555600cccfad  ; operation value
  mov r12, r11
  mov rcx, 0x2

loopstart:
  ror r11, cl          ; 32 times
  ; ...

  cmp r11, r12
  jne exit
  jmp loopstart

exit:
  push r11             ; print result
  mov rax, 0x01
  mov rdi, 0x01
  mov rsi, rsp
  mov rdx, 0x08
  syscall
  mov rax, 0x3C        ; exit
  syscall
```

Listing 4.1: Test code written in assembly to analyse if repeated executions on instable operating points cause execution erros.

Test 1 Our first test case is shown in Listing 4.1, it is rotating a constant value two positions to the right 32 times and checking the result at the end of the loop. For the first test round, we analyze the performance counters after 1 second of execution.

The test case runs multiple times. The focus of our observations lied on the level 2 cache since we expect a higher access rate from the tests which run at an undervolting level. The tests were performed once without undervolting at a frequency level of 1.6GHz and once with the same frequency at a level of -108mV. The average results of 10 executions are shown in Table 4.6.

Table 4.6: Results of the cachehammer test case running on two different undervolting levels. No significant differences can be seen.

	L1I acc.	**L1I miss**	**L1I hit**	**L2 hit**	**L2 code req.**
0mV / 1.6GHz	70494452	9377	70450645	43989	47314
-108mV / 1.6GHz	70427334	9437	70442970	44437	47065

As can be seen in Table 4.6, unfortunately, there is no measurable difference between the two averaged execution results. Also, running the test case on a more unstable operating point did not result in any noticeable changes in the execution. Next to the rotation instructions also the increment and the division instruction were tested using the same test case template as in Listing 4.1. Using these two instructions, no measurable difference in the L1I cache misses, or the level 2 cache access could be discovered.

According to the results of the performance counter, our approach of hammering the instruction cache is working. However, we do not measure a higher access rate on the level 2 cache, and therefore we conclude that bitflips in the instruction cache do not happen.

Test 2 On our second test case round, we did not stop the execution after 1 second and reviewed the performance counter. Our test cases run in an endless loop, and only a modified computation result can end the program. This test case has the purpose of verifying if instructions are fetched wrongly and produce a wrong computation result. If the test case ever ends, the result is printed and can be analyzed.

Exhaustive testing of the ror and increment instruction did not lead to any observations after several minutes of running at different undervolted levels. Either the whole system is crashing, or the test case is just executing.

The division test case showed more interesting results. After some testing, the test case crashes with a floating point exception at an undervolting level between -114 and -116mV and a frequency of 1.6GHz. This is very interesting and the first success of our approach. This exception can be triggered reproducible by the test case shown in Listing 4.2.

The exception arises between the instructions, this behavior is not printing the operation result and does not allow further analysis. To extract the result, we incrementally increase the number of divisions in the loop to see which fails. The analysis shows that the third division calculates the wrong remainder. Instead of the expected result `0xb7431ae0fe734` it calculates `0xb7431ce0fe734`. The flip from a hexadecimal A to C equals the flip of two bits.

This flip can be reproduced reliably. This finding is not what we expected to find. It is a minimal change in the result. Wrongly fetched instruction should lead to a much bigger change. The difference in the outcome is only two bits. We assume that the flip is not happening in the cache. Instead it is probably somewhere in the execution units.

```
  mov rbx, 0xa7f90c16e709c9b4  ; right part dividend
  mov rcx, 0x25f4895b54ad      ; left part dividend
  mov r8,  0xadf1232158eefa    ; divisor
  mov r11, 0x30a6ed49f6f43335  ; compare quotient
  mov r12, 0x91b75d68ccf644    ; compare remainder
  push r8

loopstart:
  mov rax, rbx
  mov rdx, rcx
  idiv qword [rsp]
  idiv qword [rsp]
  idiv qword [rsp]
  idiv qword [rsp]
  idiv qword [rsp]
  cmp rax, r11              ; compare quotient
  jne exit
  cmp rdx, r12              ; compare remainder
  jne exit
  jmp loopstart

exit:
  push rax                  ; push quotient
  push rdx                  ; push remainder
  mov rax, 0x01
  mov rdi, 0x01
  mov rsi, rsp
  mov rdx, 0x10
  syscall

  mov rax, 0x3C             ; exit
  syscall
```

Listing 4.2: Test case which triggers a floating point exception when executed on an unstable operating point. The example contains five division instructions to put higher pressure on the division unit than only with one instruction.

Chapter 5

Attack Vectors

Exhaustive testing in the previous chapter leads to a bitflip and wrong execution results using the division instruction. In this chapter, we carefully analyze the flipping behavior on the i3-7100U CPU. On this CPU, it was far easier to introduce bitflips than on our second test CPU the i5-6200U. Therefore, we mainly focused on the i3-7100U, and all results where obtained from test cases executed on it.

5.1 Vulnerable Instructions

As in the previous section, we run our test cases using the same conditions, a critical undervolt state, and a trimmed frequency to 1.6GHz. Instead of a bitflip in the instruction cache using the right instructions, it is possible to introduce bitflips in the execution units. Several instructions were analyzed of this behavior. On the division and multiplication instruction, it possible to successfully induce bitflips.

5.1.1 Division

The division instruction was the first operation, which showed a flipping behavior in our previous tests and is, therefore, our first target. The bitflip occurred in the remainder of the division, and only two bits changed. We adopt the test case shown in Listing 4.2 to only perform a single division instruction. This causes the same behavior of the result. We generated random division binaries to see if a pattern in the flipping behavior occurs.

In our automated binary generation framework, we select three values at random, which operate as our high 64bit dividend, low 64bit dividend, and a 64bit divisor. By choosing a value, it is a 50% chance that the value is from a 32bit range or from a 64bit range. This is done prior to the final value selection to verify if the flip depends on high division values (64bit).

Some of the discovered bitflips are shown in Table 5.1. The byte containing the bitflip is always byte 3 if we start counting at zero by the least significant byte. Furthermore, the flip only happens in the remainder and not in the quotient. From the results Table, it can also be seen that the flip occurs independently of the size of the divisor. Also, the size of the dividend does not influence the flipping behavior.

Table 5.1: Analysis results of division operations from random values. The Table shows the executed division and the expected and calculated result. The flip cell shows the bitflip in more detail, the "x" is representing one byte to keep the cell more readable.

`Division: ((0x37dc5d93ba7362<<64) | 0x0) / 0xadf1232158eefa`	
`Expected: 0x5236a40a88e186de & 0xb7431ae0ae734`	`Result: 0x5236a40a88e186de & 0xb7431ce0fe734`

	`Flip: 1010 1110 xxx`
	`1100 1110 xxx`

`Division: ((0x9fac0cd8<<64) | 0x8b01286974adfcd2) / 0x274aaed5ea98c8fa`	
`Expected: 0x410526c82 & 0xbc04b67de3a75de`	`Result: 0x410526c82 & 0xbc04b67fe3a75de`

	`Flip: 1101 1110 xxx`
	`1111 1110 xxx`

`Division: ((0xc2bdff09<<64) | 0xfc3e616fd7a6fad2) / 0xe7ec0267e2265d4e`	
`Expected: 0xfffffff7e97ca827 & 0x1e8a97006cb93f0`	`Result: 0xfffffff7e97ca827 & 0x1e8a97026cb93f0`

	`Flip: 0000 0110 xxx`
	`0010 0110 xxx`

`Division: ((0xd7e7a8ad<<64) | 0xc3b29e55) / 0x4d91aaef9496d6e0`	
`Expected: 0x2c88c547b & 0x1ea37a065e37e0b5`	`Result: 0x2c88c547b & 0x1ea37a065f37e0b5`

	`Flip: 0101 1110 xxx`
	`0101 1111 xxx`

`Division: ((0x50ccd07a<<64) | 0xcc6b2f9bb4b9102d) / 0xf65b414a`	
`Expected: 0x53f68393f199b5f7 & 0xce49bfc7`	`Result: 0x53f68393f199b5f7 & 0xee49bfc7`

	`Flip: 1100 1110 xxx`
	`1110 1110 xxx`

`Division: ((0xf06a14d0<<64) | 0xd18583b84aa888c7) / 0xc5cc60df9a2a77a1`	
`Expected: 0xffffffbde89806f & 0x20f1b75c934529f8`	`Result: 0xffffffbde89806f & 0x20f1b75c944529f8`

	`Flip: 1001 0011 xxx`
	`1001 0100 xxx`

The bitflip in the remainder does happen quite frequently. Out of 35 random generated binaries, 11 produce a wrong calculation result. Given us a 31.43% chance that an arbitrary division instruction fails when executed on unstable operating points. To induce flips in the remainder, an undervolt level of -114mV is required. This is a highly unstable operating point which freezes the CPU frequently.

Interestingly the flip almost always occurred in the remainder. Only in one test case, the bitflip occurred in the quotient. This sample is shown in Table 5.2. This flip occurs on an undervolting level of -116mV.

Table 5.2: Division test case which induced a bitflip in the quotient.

Division	Expected	Result	Flip

`0x35351d1d1bdee878 / 0x31afe98`	`0x1122334455`	`0x1122334456`	`0101`
			`0110`

Due to the high undervolting level needed to induce bitflips in the division instruction, not many test scenarios have been done.

5.1.2 Multiplication

The discovered bitflip in the division instruction and the finding that the flip is not happening in the instruction cache let to the assumption that probably only complex instructions are susceptible to bitflips. After the division, we tested the multiplication since it has a similar complexity.

A bitflip was already achieved by an undervolt level of -108mV and a fixed frequency of 1.6GHz. Running the test cases on this operating point, the operating system showed no noticeable changes in its behavior and ran stable compared to the division test cases. This made testing for flips way easier and could be better automated. The execution results of a tight multiplication loop were frequently wrong. Our test case can be seen in listing 5.1. The repeatedly executed multiplication instruction in a tight loop that fit in the LSD is not always returning the same result.

The loop in the assembly code is getting executed from the loop stream decoder. This means that the program is only loaded once from the instruction cache and μop-Cache. All other accesses happen from the LSD. We can verify this by using performance counters. This lets us assume that the bitflip is probably happening in the execution unit of the multiplication in port 1.

To better analyze the flipping behavior of the instruction, we built a framework that is generating the test case as in Listing 5.1 but with random multiplication values. We noticed that not all values are leading to wrong results. After careful analysis, we can say that it happens quite rarely, 1 out of 20 random multiplications produce an unexpected result interestingly, if a multiplication fails, at least every second test case execution fault. This means for further test cases that a low number of repetitions like ten should always be enough to discover a faulting multiplication. The flipping behavior of several multiplications is shown in table 5.3. Only the lower 64bit of the results are shown in the table because if a flip happened, it was in the lower half of the result.

```
  mov rdi, 0x1122334455667788  ; multiplication value
  mov r9,  0xdeadbeef          ; multiplication value
  mov rdx, 0x483726147e4887f8  ; compare value
  mov rsi, 0x3b9aca00          ; iteration max
  xor r15, r15                 ; counter register

loopstart:
  inc r15
  mov r13,r9
  imul r13,rdi
  cmp r13, rdx
  jne exit
  cmp r15, rsi
  jne loopstart

exit:
  push r15             ; print iterations
  mov rax, 0x01
  mov rdi, 0x01
  mov rsi, rsp
  mov rdx, 0x08
  syscall

  mov rax, 0x3C        ; exit
  syscall
```

Listing 5.1: Testcase which shows that the repeatable execution of a multiplication on unstable operating points lead to wrong computation results. The coorectness of the execution is verified by comparing the lower 64bit of the result. To detect a too early stop of the program the number of iterations is printed.

Table 5.3: Analysis results of multiplication of two random values. The flip column shows in the first line the expected results and the second line the computed result. The "x" in the two lines is representing one byte, the next bits are from byte 3 and in two cases also the first 4 bits of byte 4.

Multiplication	Expected	Result	Flip
0x1122334455667788 * 0xdeadbeef	0x483726147e4887f8	0x483726145e4887f8	0111 1110 xxx 0101 1110 xxx
0x8d9003867e6cb852 * 0xd6dbee7f98ecf359	0x0ae8aa1bc259ea82	0x0ae8aa1ba259ea82	1100 0010 xxx 1010 0010 xxx
0xd6dbee7f98ecf359 * 0x8d9003867e6cb852	0x0ae8aa1bc259ea82	0x0ae8aa1b4259ea82	1100 0010 xxx 0100 0010 xxx
0xf02eb72d4e1f92d6 * 0x4ceb4f0a	0x662239741dfdc65c	0x66223973ddfdc65c	0100 0001 1101 xxx 0011 1101 1101 xxx
0x6a4431d2f7009b41 * 0xf289b44e2b5d719b	0x318b20dd2382b15b	0x318b20dd1b82b15b	0010 0011 xxx 0001 1011 xxx
0xfa201ed30ca88ddf * 0x99c7f257a4fd3064	0x97c11f109dd43b1c	0x97c11f1095d43b1c	1001 1101 xxx 1001 0101 xxx
0x78f79d3d85c23931 * 0x801f750b2ae3d76d	0xec66f13dd22d80dd	0xec66f13db22d80dd	1101 0010 xxx 1011 0010 xxx
0x97ec3245 * 0x60b31670	0x3962da51c9e6ec30	0x3962da51b9e6ec30	1100 1001 xxx 1011 1001 xxx
0x4010bb9a1a4e893b * 0xedc7f221c7291ab9	0xc2fcf2e4d42429a3	0xc2fcf2e4cc2429a3	1101 0100 xxx 1100 1100 xxx
0x94225b5d * 0xfa093c0c	0x90aee52d074b145c	0x90aee52cc74b145c	1101 0000 0111 xxx 1100 1100 0111 xxx
0xa9497e39 * 0xfaead8af	0xa5ed11bac5d760f7	0xa5ed11ba45d760f7	1100 0101 xxx 0100 0101 xxx

As the result table shows, if a bitflip happens, it is either in the third or fourth byte when we start counting from the least significant bit as byte 0. Exhaustive testing was done by generating more than 100 multiplications. The random multiplication operands where chosen from a 64-bit and 32-bit range to verify if the fault depends on the size of the operands. As can be seen in the table, no pattern is detectable in the flipping behavior.

Our first test case used the multiplication instruction ”imul” of the form ”mnemonic op1, op2”. To verify if the flipping behavior is depending on a specific instruction mnemonic, we tested other types of multiplication instructions. We listed the tested instructions in Table 5.4.

Table 5.4: Result of testing different instruction forms to verify if the bitflip is depending on a specific mnemonic type.

Instruction Type	*µops*	**Flips**
imul r64	2	yes
imul r64, r64	1	yes
imul m64	3	yes
imul r64, m64	2	yes
imul r64, m64, imm32	2	yes

As can be seen in the results table, the bitflips are not depending on the instruction type. It is also not depending on the number of *µops* it is getting decoded to. Testing different types of multiplication with the same values lead to the same bitflips in the result. This lets us further conclude that the bitflip may come from the execution unit.

It does not depend on the instruction type. However, we noticed during our tests that the occurrence of bitflips strongly depends on the multiplication values. To compare the results shown in Table 5.3, we also build a table of multiplication values that did not end up with any bitflips. Some of these are shown in Table 5.5. Since most flips are happening in the third byte, we added these bits to the table to get a more clear view.

Table 5.5: Results of the analysis of random multiplications which do not produce any bitflips.

Multiplication	**Result**	**Bits**
`0x557bf4ae * 0x955b90e141db01cd`	`0x316b87b26f119d56`	`0110 1111 xxx`
`0xb56d9f52 * 0xa1c1ad18`	`0x72a33383c5c359b0`	`1100 0101 xxx`
`0xd5a0cf9f82babcf8 * 0x809e0454bcefea30`	`0x7724cbe3bf461e80`	`1011 1111 xxx`
`0x77c2ec87289571c5 * 0x5abe5e02`	`0x92f422dee327398a`	`1110 0011 xxx`
`0xaf5cf5c1d6a03139 * 0x6a78383111804b85`	`0xe39d4edbd525459d`	`1101 0101 xxx`
`0x7b86b42d * 0x31411c73ef13c336`	`0x0901279e0bff487e`	`0000 1011 xxx`

It is not visible why these multiplications do not produce wrong results since there are similar bit patterns in byte 3 compared to Table 5.3. To further test when a bit flip is occurring and when not, we recreated several bit patterns from the flipping test cases and reproduced those patterns by using different multiplication values. These results are shown in Table 5.6, the analyzed pattern is surrounded by spaces.

Table 5.6: Analysis of bitflips depending on various bit patterns. Each pattern was tested by ten different multiplications running a single multiplication ten times.

Multiplication	Result	Flips
0xafded2a8f58b5d88 * 0x3a7269f8	0xdb224ba 47e ef63c0	-
0xcd5b1eb0aea061e4 * 0xbb36ab0b0abca2e5	0xa0e044f 47e d9d8f4	-
0xafded2a8f58b5d88 * 0x3a7269f8	0xf0248e4 47e 8c2a7d	1
	Summary	1/10
0x76b71761e89a5ab1 * 0x6c0c8955	0xcd062e7 bc2 14d5c5	-
0x06b342a11ccbced9 * 0xd92a6110332b13ed	0xb26a3b4 bc2 7b99e5	0xb26a3b4bbe7b99e5
0x351f2032 * 0xb84d5ec9	0x263e6d2 bc2 4ca342	-
	Summary	1/10
0xc31d7077cd6f59c9 * 0xa15a9350	0xc319310 41d 0479d0	-
0xbc4cb8c7 * 0xbdf79eff	0x8bbac76 41d 77e039	-
0x7a9da95463235e97 * 0x4e05b3bcfa1b574a	0xaee5ab1 41d 4ba8a6	-
	Summary	0/10
0xa377723d6b18530 * 0x677526c3dd8fcfaf	0x648a2cc d23 dbdbd0	-
0x8f7d0023321ee1b * 0x83b7c2c7d4adf502	0x1bf3ef0 d23 62b336	1
0xfa500aa5 * 0x58a51eac	0x56acf73 d23 5f7cdc	-
	Summary	1/10
0xf0a5645b * 0x92a61c8dd8aba38b	0x66c11f0 1c9 7c6e69	-
0x581c8389 * 0x795a34d7	0x29c4866 1c9 d44c0f	-
0x46d155f729801925 * 0xae3ce0e78486c506	0xc4180d2 1c9 b80fde	-
	Summary	0/10

The result of ten different multiplications producing a bit pattern, which regularly flipped using the original multiplication, showed that there is no dependence between a pattern in byte 3 and 4 of the computation result and the flipping behavior. If there was a dependency, we should have seen far more flips on equal bit patterns.

The patterns from the previous test cases did not show any coherent information. Trying to introduce a dependency between the flips and a pattern, we tried more uniform patterns and analyzed the behavior. The analysis of two 12 bit patterns is shown in Table 5.7.

Table 5.7: Results of specific bit patterns in the flipping bytes.

Multiplication	Result	Flips
0x8d59591c28b24efd * 0x37676eb8	0xe394d47000e47bd8	-
0x1c4c967ae60f40d4 * 0x52b867fd	0x4e4267c000885d84	-
0x90effa0a1c3d3e67 * 0xb87ab894777a5bcd	0x5756dae0004f957b	-
	Summary	0/10
0x19825252 * 0xed6283fe6bbaa4cd	0x778db4dfffac73aa	1
0xcbaca291 * 0x92ba44f3	0x74bca6ffff66d3a3	-
0x4736313ddba9b6be * 0x4d76efbd	0x789066efff7b4c46	1
	Summary	2/10

An extension to Table 5.7 can be seen in Table 5.8. We also tested 16-bit patterns on byte 3 and 4 with different alternating structure. The execution scenario is still the same by running each bit pattern by ten different multiplications and executing each multiplication ten times.

Table 5.8: Analysis of flipping behavior by producing specific bit patterns in the third and fourth byte.

Pattern	Binary	Flips
0x1111	0001 0001 0001 0001	0/10
0x2222	0010 0010 0010 0010	1/10
0xaaaa	1010 1010 1010 1010	2/10
0x3333	0011 0011 0011 0011	0/10
0xcccc	1100 1100 1100 1100	1/10
0x0000	0000 0000 0000 0000	0/10
0xffff	1111 1111 1111 1111	1/10

All patterns chosen followed an alternating scheme in the expectation that these patterns flip more likely, but unfortunately, after ten test runs, this did not happen. A pattern that causes more bitflips has not been found.

After the previous studies, we can assure that the flips are not depending on the bit pattern, but we strongly noticed that they are definitely depended on the multiplication values. We tried to find a pattern in these values. Our approach was to count the number of one bits in it. A high number of ones in both values may lead to extra steps during the multiplication if we recall a simple binary multiplication scheme.

We analyzed the flips from Table 5.3 and Table 5.5, but taking a closer look and counting the number of ones does not show any relation. For example, the multiplication `0x97ec3245*0x60b31670` is leading to a bitflip in the 3rd byte and has only 29 high bits. Whereas the numbers `0xaf5cf5c1d6a03139*0x6a78383111804b85` have a total of 57 ones and do not lead to a bitflip at all. Our studies also showed that there is no difference if the multiplication values are 32 or 64 bit. This lets us conclude that it does not depend on the number of ones in a variable.

After exhaustive testing, we could not identify any pattern between the multiplication values, the bit patterns, and the flipping behavior.

As a last test scenario, we analyzed what are the smallest multiplication operands causing a bitflip. We modified our framework and incrementally increased one value and choose the second one at random. We show our results in Table 5.9.

Table 5.9: Analysis of the smallest operands which cause a bitflip.

Multiplications
`0x02 * 0xf9748888e4ada8b5`
`0x09 * 0x175b33f5`
`0x0a * 0xb222736c`
`0x0b * 0xccaec987`
`0x0c * 0x82e7421d4e182cbf`
`0x14 * 0xaab2f85c`

The smallest operand causing a bitflip is two. Increasing one operand shows that the result fails on multiple small operands. Interestingly, switching the operands lead to no bitflips. The bitflips on small operands can be interesting on array access. Producing a flip on an array index multiplication can lead to out of bound accesses and produce undefined behavior.

5.2 Attacking Cryptography

As Adrian Tang et al. [69] show in the CLKscrew paper by precisely undervolting and frequency scaling a CPU core close to its functional boundary, faults can be induced, and even secret keys can be leaked from attacking the ARM TrustZone.

We show in the previous sections that by using the same approach, we can influence the output of multiplication and division instructions on a regular i3-7100U. Because of this weakness in the instruction, we focus on asymmetric cryptographic algorithms since these are based on modular exponentiation, which contains a lot of multiplications.

High reliance on multiplication gives us a good attack surface and a high likelihood of successfully inducing faults. This has been shown to be very critical for RSA-CRT[5], which can lead to a complete compromise of the scheme.

5.2.1 Bellcore Attack

The first work of using fault injection attacks to extract secret keys from cryptographic algorithm was done by the Bellcore research team [4]. Sidorenko et al. [1] analyzed the practical aspects of fault attacks on RSA with the focus on the CRT variant. This scheme requires only one faulty generated signature to be entirely broken. This is a powerful attack since a specific location of the fault is not required.

Given a correct signature s and a faulty signature s', an attacker can calculate the private prime q by computing the greatest common divisor of $s - s'$ and the modulus n as shown in Equation 5.1.

$$q = gcd(s - s',\ n) \tag{5.1}$$

This computation of q is possible because of the design scheme of RSA-CRT. Combing the equations from Section 2.8 shows why this attack is possible. The calculation of an RSA-CRT signature can be combined to Equation 5.2.

$$s = [(c_p - c_q) \cdot qInv\ mod\ p] \cdot q\ + c_q \tag{5.2}$$

By a successful undervolt attack, an error in one of the multiplications inside the squared brackets happened. Resulting in Equation 5.3.

$$s' = [erronous] \cdot q\ + c_q \tag{5.3}$$

Subtracting these two equations leads to the cancellation of c_q. Further simplification leads to Equation 5.4.

$$s - s' = ([(c_p - c_q) \cdot qInv\ mod\ p] - [erronous]) \cdot q \tag{5.4}$$

A successful bitflip in the computation of one signature and the subtraction of a correct one leads to a multiplication with the private prime q and an unknown factor. As well the public modulus is a multiplication with p and q. Applying the greatest common divisor as in Equation 5.1 returns q since it is the greatest factor in both values.

The Bellcore Attack is also working on classical RSA [1], but several faulty signatures are required. This fact makes this attack even more powerful.

Bellcore Attack using Undervolting

RSA-CRT is highly based on multiplications. Using our findings from Section 5.1.2, we have a high likelihood of successfully introducing a bitflip in one of the multiplications and applying the Bellcore Attack. We implemented an RSA-CRT signature scheme in C using the big integer library from GNU called GNU Multiple Precision Arithmetic Library (GMP) [22].

Our test cases showed that due to the many multiplications performed, we are not required to go close to the limit of unstable undervolting. A level of -104mV and the frequency fixed to 1.6GHz was enough to generate several faulty signatures. Each of them could be used to calculate back the private exponent p. This attack works because, in the CRT variant, both primes are operated separately on the message block and combined later. If now one operation induces a fault, the other prime can be calculated by the greatest common divisor.

The implementation of the attack is listed in Appendix A, the complete signature generation code is shown in Listing A.1 and also the corresponding attack written in Listing

A.2 using one of the faulty generated signatures.

The attack was performed in a real-world scenario where the victim does run a signature generation service. No further interaction from the victim is required. The attack was performed on non-enclave code because the i3-7100U did not support SGX. However, the attack works the same on an SGX supportive CPU. The attacker has to undervolt the system to -104mV and keep the frequency constant on 1.6GHz to reliably produce faults.

We requested the signature generation service ten times under different undervolting levels. The results can be seen in Table 5.10. A higher undervolting level leads to more faults. Interestingly every occurred fault could be used to generate a private prime. This gives us a success rate of 100% on a fault.

Table 5.10: Behavior analysis results of requesting the signature generator under different undervolting levels.

Undervolt level [mV]	Faults	Success rate
-104	2/10	2/2
-105	5/10	5/5
-106	5/10	5/5
-107	7/10	7/7
-108	7/10	7/7

Improving Bellcore Attack using Frequency Jumping

It is possible to improve the attack further and run the operating system on a more stable point. This can be achieved by decreasing the undervolting level and alternate the frequency between the maximum 2.4GHz and 1.6GHz.

Our analysis showed that when we are operating on a stable undervolting point that additional frequency jumping can cause further instability to the operating system. Running the test case on our i5-6200U CPU at an undervolt level of -138mV shows no noticeable effects on the system. Adding fast frequency scaling to this operating point crashes our test case with a segfault in the GMP library. Unfortunately, we were not able to generate faulty signatures using this method of instability on the i5-6200U.

While we are only getting crashes for the i5-6200U CPU, we were able to generate flips on a lower undervolting level on the i3-7100U. The Bellcore Attack from Section 5.2.1 was performed at an undervolting level between -104 and -108mV. Using a fast alternating frequency between 1.6GHz and 2.4GHz, we were able to get faulty signatures at an undervolting level of -90mV. This shows that it is possible to increase further instability of a system using alternating frequency levels.

In this test case scenario, we run the signature algorithm in an endless loop to see if a bitflip ever occurs. We do not care about how many flips occur. Our goal is to induce a bitflip on a more stable undervolting level.

RSA-CRT in Open Source Libraries

Besides attacking our own implementation of RSA-CRT, we also tested two open-source cryptographic libraries, namely mbed TLS and OpenSSL.

mbed TLS has a safety mechanism implemented to avoid flipping or glitching attacks, to our regret. The mitigation consists of double computing the modular exponentiation, and if the results are not equivalent, an error is thrown. This makes a calculation of the private exponent infeasible. This mitigation comes with a performance and energy trade-off, but it prevents all glitching and flipping attacks. Since producing two times, the same bitflip in two sequential multiplications is highly unlikely. This mitigation is implemented since version 2.8.0 which was released on 2018-03-16.

OpenSSL also contains a mitigation against glitching attacks by verifying the result of an RSA-CRT computation. This is implemented since the update from April 2001. This is a consequence of the work of Boneh et al. [6] on the importance of eliminating errors in cryptographic applications.

5.3 Exact Timing of Bitflip

The Bellcore Attack from Section 5.2.1 showed that bitflips in the computation of a signature could have a serious impact on the security of a cryptographic system. To perform the previous attack, it does not matter where in the operation the bitflip occurs. The attack is based on a single faulty generated RSA-CRT signature.

In this section, we show how it is possible to trigger a bitflip only on a specific multiplication instruction instead of running the system on an unstable operating point all the time.

Our current test cases are all based on the same scheme, a multiplication hotspot that fits in the Loop Stream Detector. We want to stretch these limits and test if the bitflip is depending on where the instruction is fetched and if a multiplication hotspot is really needed. Our test case, shown in Listing 5.2, contains a single multiplication which is surrounded by add instructions. For each test run, we increase the number of additions and check for wrong computation results.

Table 5.11: Testing the dependencies of a single multiplication and it's flipping behavior.

Loop size	**Issued from**	**Flip**
20	LSD	yes
60	LSD	yes
100	μop-Cache	yes
140	μop-Cache	yes

In Table 5.11, it can be seen that increasing the padding around a multiplication does not influence the flipping behavior we showed in earlier test cases. This shows that the bitflip does not depend from where the instruction is issued.

Previously, on a flipping multiplication, we had a success rate of about 50%. Targeting a single bitflip by padding the instruction reduces the success rate to 20%. This can be explained because we are now trying to produce a flip in a specific multiplication, which is far less frequently executed than in previous test cases.

Taking these results, we are able to produce bitflips with precise timing on specific locations. Giving us the opportunity to build more novel attacks on cryptographic schemes.

5.4 mbed TLS

Mbed TLS [3] is an easy to use and widely known cryptographic library. All our performed tests in this section are done on version 2.16.3. As mentioned in Section 2.8, the sliding window exponentiation method is used in mbed TLS for the modular exponent operation.

Taking a closer look at Step 3.1 from Algorithm 1, we can see that if the i-th bit in the exponent is zero, a multiplication is performed. Using a precisely timed undervolt attack, we show that it is possible to target this multiplication and can, therefore, verify if the bit in the exponent is zero or not. This results in a leak of parts from the private exponent.

Because the RSA-CRT is prone to fault attacks, it is not used in real-world anymore. By default, mbed TLS is using RSA-CRT due to its faster computations, although it can be easily disabled and switched to none-CRT mode by a configuration flag in ”config.h”.

The implementation of the sliding window exponentiation is in ”library/bignum.c”, the function itself is called ”mbedtls_mpi_exp_mod”. The part we are going to attack is shown in Listing 5.3, in the unmodified library, the multiplication is in line 1995.

For this example, we take the RSA signature program from the examples folder (”programs/pkey/rsa_sign.c”). For an RSA private exponent d as shown below.

```
d = 0x589552BB4F2F023ADDDD5586D0C8FD857512D82080436678D07F984A29D89
2D31F1F7000FC5A39A0F73E27D885E47249A4148C8A5653EF69F91F8F736BA9F848
41C2D99CD8C24DE8B72B5C9BE0EDBE23F93D731749FEA9CFB4A48DD2B7F35A2703E
74AA2D4DB7DE9CEEA7D763AF0ADA7AC176C4E9A22C4CDA65CEC0C65964401
```

Analysis of the sliding window implementation showed that the multiplication we want to target is called 140 times, giving us a possible chance of leaking 140 bits of a 1024 bit private exponent. This would be a 13.67% leakage of the secret, reducing a brute force attack from 2^{1024} to 2^{884} possibilities. Analyzing the call of the multiplication using ten different secrets lead to an average call rate of 157 times per signature calculation. This is an average leak of 15.33%.

The attack was only performed theoretically by a proof of concept, to see how many possible targets we have. The flip is simulated by a rewrite of the code.

5.5 Countermeasures

We showed that taking advantage of the detected bitflips leads to a modified execution inside an SGX enclave. Modifying cryptographic systems allowed us to extract a private RSA prime from inside the SGX memory. This is a validation of the security guarantees and needs to be fixed for future systems to reassure the security of SGX. In this section, we explain how possible mitigations for the discovered bitflips may look like. We discuss countermeasures from the software level as well as from the hardware.

Microcode update

A straightforward approach is to disable access to the MSR 0x150 by a microcode update. This is only possible if the *wrmsr* instruction, which handles the write to the register, is decoded via the microcode sequencer. If this is the case, the decoding of the instruction can result in no operations if the specified register is 0x150. This would prevent an attacker from moving the system into an unstable operating point and mitigate this attack vector. However, undervolting is frequently used to reduce the power consumption of a CPU by many hobbyists and professionals. Also, this approach does not target the root cause of bitflips. Unstable operating points might still be achieved by targeting the overall supply voltage through other power or clock management features, which might be implemented in the future.

Hardware mitigations

A hardware approach could be to define operating points in the hardware. Giving software the chance to perform undervolting, but if the assigned level is outside a manufacturer defined boundary, the hardware does not apply it. This provides the software with free operating possibilities, but the hardware is watching and behaving like a shield to prevent unstable operating points. This would require a hardware change and is, therefore, more expensive to realize than a microcode update.

Another hardware mitigation could be to execute the SGX code twice and compare the result. This is a common countermeasure against fault and glitching attacks on cryptographic systems. It is also implemented in lockstep CPUs [59] used for critical infrastructures where every failure can lead to catastrophic effects. The double calculation leads to overhead in the execution and extra die size by an extra hardware unit. Next to an extra hardware implementation, the result can also be verified by executing the code on a different core or in a separate hyper thread.

Software mitigations

A software mitigation by running the same program on different cores or hyper threads is a waste of resources depending on how computationally intensive the program is. The two tested open-source libraries openSSL, and mbed TLS already have builtin mitigations against bitflips. Both libraries recalculate results of cryptographic algorithms, which are known to be susceptible to bitflips.

The proof of concept attack on mbed TLS shown in Section 5.4 can be mitigated by an already implemented feature called exponent blinding [65]. By adding a random multiple of Euler's ϕ function of the modulus to the private exponent, our attack is no longer able to extract the secret prime. Instead, the extracted value is the blinded exponent. This feature has a slight impact on the performance due to the extra multiplications needed for encryption and decryption. In mbed TLS, exponent blinding is implemented since version 2.5.0. It was added to prevent side-channel attacks on RSA [67]. This mitigation changes the private prime before the sliding window exponentiation and renders our attack useless.

```
  mov rdi, 0x1122334455667788  ; multiplication value
  mov r9,  0xdeadbeef          ; multiplication value
  mov rdx, 0x483726147e4887f8  ; compare value
  mov rsi, 0x3b9aca00          ; iteration max
  xor r15, r15                 ; counter register

loopstart:
  add rbx, 1
  ; ...

  mov r13,r9
  imul r13,rdi
  cmp r13, rdx
  jne exit

  add rbx, 1
  ; ...

  inc r15
  cmp r15, rsi
  jne loopstart

exit:
  push r15              ; print iterations
  mov rax, 0x01
  mov rdi, 0x01
  mov rsi, rsp
  mov rdx, 0x08
  syscall

  mov rax, 0x3C         ; exit
  syscall
```

Listing 5.2: Testcase to verify if the generated bitflips are depending from where an instruction is fetched and if a multiplication hotspot is needed or not.

```
1984 /*
1985  * skip leading 0s
1986  */
1987 if( ei == 0 && state == 0 )
1988   continue;
1989
1990 if( ei == 0 && state == 1 )
1991 {
1992   /*
1993    * out of window, square X
1994    */
1995   MBEDTLS_MPI_CHK( mpi_montmul( X, X, N, mm, &T ) );
1996   continue;
1997 }
1998
1999 /*
2000  * add ei to current window
2001  */
2002 state = 2;
2003
2004 nbits++;
2005 wbits |= ( ei << ( wsize - nbits ) );
2006
2007 if( nbits == wsize )
2008 {
2009   /*
2010    * X = X^wsize R^-1 mod N
2011    */
```

Listing 5.3: Code cutout from the *mbedtls_mpi_exp_mod* function in mbed TLS. The performed montgomery multiplication in line 1995 is depending on *ei* which is a single bit in the exponent. The *state* variable describes the sliding window state from the multiplication.

Chapter 6

Related Work

Previous work on fault attacks is mainly based on hardware faults induced by physical access to the device. A good summary of attacks has been done by Karaklaji et al. [38]. While the listed attacks need physical access to the device, there are more novel attacks where remote access on the target is enough. The first work which had the idea of stretching the combinatorial delay without physical access to the device was CLKscrew [69] in 2017. This research provides the prerequisites for all of the following work, including this thesis. CLKscrew targets the ARM Trust Zone of mobile devices. Their achievement includes extraction of AES keys by inducing targeted bitflips from remote.

During the work of this thesis, three papers [51, 40, 61] were published with the same approach and ideas. All are leading to the same result, inducing a bitflip in the execution of a program running inside SGX.

V0ltpwn by Kenjar et al. [40], Plundervolt by Murdock et al. [51], and Voltjockey by Qiu et al. [61] used the work from Tang et al. [69] and applied it on Intel CPUs to target SGX. Stretching the combinatorial delay by modifying the frequency and the supply voltage, faults in the execution could be injected, leading to a novel attack that can be applied remotely. The vulnerability was disclosed to Intel and got assigned CVE-2019-11157.

The test case scenarios from V0ltpwn use vector instructions, single instruction multiple data (SIMD). The bitflips they discovered did not happen in the multiplication or division unit, as we discovered. Instead, their flips occurred in a SIMD memory transfer. The V0ltpwn paper also mentions that during their work, they recorded invalid instructions. This might arise either from errors in the decoder or in the instruction cache. These results suggest that our initial approach of inducing bitflips in the instruction cache could work on a different CPU.

While V0ltpwn is attacking SIMD instructions, Plundervolt attacks single data instructions (SISD) like multiplication. Voltjockey does not describe which kind of instructions they target, but they could successfully attack an AES implementation. It is not explained if the attack happens on a custom AES implementation, an open-source implementation, or by using AES-NI instructions. Plundervolt achieved bitflips in AES-NI, leading to the extraction of the AES key inside the enclave. Furthermore, Plundervolt showed by a proof of concept that faulting a pointer multiplication can redirect the write of a secret value outside of the enclave to non-enclave memory. After this publication, we also tried to fault AES-NI instructions on our i3-7100U. Unfortunately, we did not achieve a bitflip at several different undervolting levels.

After the disclosure of these works, Intel's mitigation plan is to disallow access to the MSR 0x150, also called the overclocking mailbox interface.

Chapter 7

Conclusion

In this thesis, we presented an in detail analysis of the effects on running CPUs on a critically low voltage and a scaled frequency. Furthermore, we made a detailed analysis of the functionality from the μop-Cache. We carefully analyzed the decoding and storage behavior of several instructions in relation to the μop-Cache. We found interesting results on some instructions which were not expected to find in the beginning. We found that the fetch of a single instruction depends on the immediate value. Different values are leading to fetch from either the μop-Cache or the instruction cache. This behavior is not explainable to us and should be analyzed further in future work.

The CacheHammer attack performed in this thesis showed that putting the instruction cache under high pressure does not result in any observable delay. The analysis of several instructions did not show any result, so we came to the conclusion that this approach is not working as we were expecting. However, during the analysis, we detected a bitflip in the execution units of the multiplication and division instruction. With this bitflip, we could successfully exploit cryptographic algorithms running inside an SGX enclave. Attacks have been performed on a self-implemented RSA-CRT signature scheme and on the sliding window exponentiation implementation of mbed TLS.

We examined countermeasures to mitigate the bitflips. The easiest mitigation to avoid all flips is to prevent access to the MSR 0x150. More sophisticated mitigations were discussed, including hardware changes.

Analyzing the exploitability of Intel CPUs under low voltage supply was an open research question and has been shown to be true and exploitable.

Appendix A

Bellcore Attack

```c
#include <gmp.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
  mpz_t const1; mpz_init(const1); mpz_set_str(const1, "1", 10);
  mpz_t e; mpz_init(e); mpz_set_str(e, "65537", 10); // public exponent e

  mpz_t p; mpz_init(p); // private prime p
  mpz_set_str(p, "bf9fd0f2e7713b4b9ddc2d3b2495c54727d19341e7a46bf93361e44d361e
  0f91a7fe81a4cd18d3efe45fff1a22f667a38589a52a198a41f685b51a25dc8bd214db5f3f1b
  50bc8f7b4d8249283c919115c46a5caff43f374376a80e50065b0d519c9279c660161db9b204
  b1fbb2e35d83e94cc951f3e91af5502ca9d6ad7cede3", 16);

  mpz_t q; mpz_init(q); // private prime q
  mpz_set_str (q, "cf28842fd288d2e251bfc0cfc2c52a548a8ff7cb3dbadbdf370f68cef5a
  82868a069c9147252051b3dc63198fc62b6b8adf0e96de90a32d65e4684e28d6861b3fc52749
  9878d5aba11051ae0ab01642934be4713cebe65a49b14e184b50d3060003cfb0e6300b56ae63
  1f72a805d8936b78e5884e4ce52e9c39ac33410000c47", 16);

  // calculate public modulus
  mpz_t n; mpz_init(n); mpz_mul(n, p, q);

  mpz_t p1; mpz_init(p1); mpz_sub(p1, p, const1);
  mpz_t q1; mpz_init(q1); mpz_sub(q1, q, const1);
  mpz_t phi; mpz_init(phi); mpz_mul(phi, p1, q1);

  // calculate private exponent d
  mpz_t d; mpz_init(d); mpz_invert(d, e, phi);

  // perform CRT steps
  mpz_t dp; mpz_init(dp); mpz_mod(dp, d, p1);
  mpz_t dq; mpz_init(dq); mpz_mod(dq, d, q1);
  mpz_t qinv; mpz_init(qinv); mpz_invert(qinv, q, p);

  // sign message "1234567890"
  mpz_t m; mpz_init(m); mpz_set_str (m, "1234567890", 10);

  mpz_t sp; mpz_init(sp); mpz_powm(sp, m, dp, p);
  mpz_t sq; mpz_init(sq); mpz_powm(sq, m, dq, q);

  mpz_t s_tmp; mpz_init(s_tmp); mpz_sub(s_tmp, sp, sq);

  mpz_t h; mpz_init(h);
  mpz_mul(h, s_tmp, qinv);
  mpz_mod(h, h, p);

  // calculate final signature
  mpz_t s; mpz_init(s); mpz_mul(s, h, q); mpz_add(s, sq, s);
  printf("Signature: "); mpz_out_str (stdout, 16, s); printf("\n");

  // verification
  mpz_t md; mpz_init(md); mpz_powm(md, s, e, n);

  if(mpz_cmpabs(m, md) != 0) { printf("Verification Failed\n"); }
  else { printf("Correct\n"); }
  return 0;
}
```

Listing A.1: Complete C code to create a RSA-CRT signature of the message ”1234567890” using the GMP library. This code produces different signatures if it is running on unstable operating points.

```python
from Crypto.Util.number import GCD

# modulus n
n = 0x9b108ddf42c4d8bb0d673c054406ce366c96ac49addde59e5bfd441e2fd22d3a66268
4a267bed25f948c5a50e0b51b085f728d28ca24cd140365bc6c18960a12cd0459e2d07fb803
6d51547d29bfdee47caf418b1247012f4ec091cfd3f7ad7cb23a56a4cd6239c0a2b13557303
854b9c7c427745941b588589edf31804478c27bd7a4740ab5ab5bf73a6746219049d35ad5dc
1de5ca7b68000b30b90f48b2b436afa5d5e47d4805d14a83e27770bde09cd09e33d40621643
658172de370014c5755fb36fa5b3c6df22a16b4108c07cd8a54639ba0ba6ec8b61c5ad45880
f29f4b1798205e2b2cf40c36eb507f4b7f055eebbc636b398a20295334a628cc9df5

# correct signature
sc = 0x9063de5a24d04602e70c9442e987b44b177fd3219db832418a814080ff76b100981d
5abb971118afe90d67a45116c75a6afb58fa625cab0b65fe72bb0766073010ab60a2bc4fc66
6834712299e2217700499bb5892522e7ce2790e4583f1587ec9d281611e0913116fe82fb56e
d1ff2bde67955acde04cf385291f366ab11ae596dbb22775fcf7eafeab703584af5ac3524e4
af3f5e523771e6f60fcced6fd6562e17d0b5b20c13015b4b47de1607b64c235fc914a253149
2318e9f0c154c6283f62209a9f215d48eca0d48ff2dbac35229a451dd449ed8e8b0c1de7147
69da16b4b785f3851da745cc0d1883684e09443028d199e5b6afd722e27aa1dabda76

# faulty signature
sf = 0x8eb1148c900b0965a9e32ed6898e620b9ade40652dbb98c366ba4b675c297e2eb1b7
a0a7b53a45b8b7e5518b7bc9b81c26e952d74a700cf01611aed7dd9246f227899cf351deae0
2440a3ad98e38ddb9e6e9207b0060e93df890bef7f0a1321cccc78b9068ffdb517a3c6ac1fc
2717f207dc36d08b9a8c6f996028cf4bfbb19f80ed972152529aad5e1be7c993c775963795a
fb7722c031ee63dff46734fdebbea24a748a19e08523aa86106534b342d4c6d0e67f138f384
ead1e07fa3d763db0e72e737f3c08d752ff1f21271737ba07d2a7d65aa691f40c18d6e4fc99
3210a6d8b6cd4f111bd38780d1113f7e06f2c7e6904e0179c4eeaf303d70e4ccb2282

prime_p = GCD((sc-sf) % n, n)
print("Prime P: {}".format(prime_p))

# prime_p:
# 0xbf9fd0f2e7713b4b9ddc2d3b2495c54727d19341e7a46bf93361e44d361e0f91a7fe81a
# 4cd18d3efe45fff1a22f667a38589a52a198a41f685b51a25dc8bd214db5f3f1b50bc8f7b
# 4d8249283c919115c46a5caff43f374376a80e50065b0d519c9279c660161db9b204b1fbb
# 2e35d83e94cc951f3e91af5502ca9d6ad7cede3
```

Listing A.2: Bellcore attack on the previous implemented signature scheme in python, the output of the attack is the private exponent.

Bibliography

[1] Andrey Sidorenko, Joachim van den Berg, R. F. M. G., and de Vos, J. Bellcore attack in practice. In *IACR Cryptology ePrint Archive* (2012).

[2] Apecechea, G. I., Eisenbarth, T., and Sunar, B. Mascat: Stopping microarchitectural attacks before execution. *IACR Cryptology ePrint Archive 2016* (2016), 1196.

[3] ARM. mbed TLS. `https://tls.mbed.org/`. Retrieved on Dezember 8, 2019.

[4] Berzati, A., Canovas, C., and Goubin, L. Perturbating rsa public keys: An improved attack. In *Cryptographic Hardware and Embedded Systems – CHES 2008* (Berlin, Heidelberg, 2008), E. Oswald and P. Rohatgi, Eds., Springer Berlin Heidelberg, pp. 380–395.

[5] Boneh, D., DeMillo, R. A., and Lipton, R. J. On the importance of checking cryptographic protocols for faults. In *Advances in Cryptology — EUROCRYPT '97* (Berlin, Heidelberg, 1997), W. Fumy, Ed., Springer Berlin Heidelberg, pp. 37–51.

[6] Boneh, D., DeMillo, R. A., and Lipton, R. J. On the importance of eliminating errors in c ryptographic computations. *Journal of Cryptology 14*, 2 (Mar 2001), 101–119.

[7] Bristol, C. Describe the binary, m-ary and sliding window exponentiation algorithms. `http://bristolcrypto.blogspot.com/2015/02/52-things-number-24-describe-binary-m.html`. Retrieved on Dezember 8, 2019.

[8] Burton, E. A., Schrom, G., Paillet, F., Douglas, J., Lambert, W. J., Radhakrishnan, K., and Hill, M. J. Fivr — fully integrated voltage regulators on 4th generation intel® core^{TM} socs. In *2014 IEEE Applied Power Electronics Conference and Exposition - APEC 2014* (March 2014), pp. 432–439.

[9] Chen, D. D., and Ahn., G.-J. Security analysis of x86 processor microcode.

[10] Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., and Lai, T. H. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. *2019 IEEE European Symposium on Security and Privacy (EuroSP)* (Jun 2019).

[11] Chiappetta, M., Savas, E., and Yilmaz, C. Real time detection of cache-based side-channel attacks using hardware performance counters. *Appl. Soft Comput. 49*, C (Dec. 2016), 1162–1174.

[12] College, E. Pentium FDIV bug. `https://www.cs.earlham.edu/~dusko/cs63/fdiv.html`. Retrieved on January 30, 2020.

[13] Costan, V., and Devadas, S. Intel sgx explained.

[14] Crowe, J., and Hayes-Gill, B. 6 - flip-flops and flip-flop based circuits. In *Introduction to Digital Electronics*, J. Crowe and B. Hayes-Gill, Eds. Newnes, Oxford, 1998, pp. 150 – 163.

[15] ElectronicNotes. SRAM Memory. `https://www.electronics-notes.com/articles/electronic_components/semiconductor-ic-memory/static-ram-sram.php`, 2019. Retrieved on November 19, 2019.

[16] Eleršič, M. linux-intel-undervolt. `https://github.com/mihic/linux-intel-undervolt`, 2018.

[17] Ferguson, N., and Schneier, B. *Practical Cryptography*, 1 ed. John Wiley Sons, Inc., USA, 2003.

[18] Fog, A. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly pro-*

grammers and compiler makers, 2016.

[19] Gadhe, A., and Shirode, U. Read stability and write ability analysis of different sram cell structures.

[20] Giraud, B., and Amara, A. Read stability and write ability tradeoff for 6t sram cells in double-gate cmos. In *4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008)* (Jan 2008), pp. 201–204.

[21] Glynn, K. ThrottleStop. `https://www.techpowerup.com/download/techpowerup-throttlestop/`, 2019. Retrieved on November 26, 2019.

[22] GNU. The GNU Multiple Precision Arithmetic Library. `https://gmplib.org/`, 2016. Retrieved on November 12, 2019.

[23] Gruss, D., Maurice, C., and Mangard, S. Rowhammer.js: A remote software-induced fault attack in javascript. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721* (Berlin, Heidelberg, 2016), DIMVA 2016, Springer-Verlag, p. 300–321.

[24] Gueron, S. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Report 2016/204, 2016. `https://eprint.iacr.org/2016/204`.

[25] Götzfried, J., Eckert, M., Schinzel, S., and Müller, T. Cache attacks on intel sgx. pp. 1–6.

[26] Hill, M. D., and Smith, A. J. Evaluating associativity in CPU caches. *IEEE Transactions on Computers* (1989), 1612–1630.

[27] Horowitz, M. 1.1 computing's energy problem (and what we can do about it). *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)* (2014), 10–14.

[28] Intel. Power Management States: P-States, C-States, and Package C-States. `https://software.intel.com/en-us/articles/power-management-states-p-states-c-states-and-package-c-states`, 2014. Retrieved on November 25, 2019.

[29] Intel. PACKAGE AND PLATFORM VIEW OF INTEL'S FULLY INTEGRATED VOLTAGE REGULATORS (FIVR). `https://www.psma.com/sites/default/files/uploads/tech-forums-packaging/presentations/is87-package-and-platform-view-intel%E2%80%99s-fully-integrated-coltage-regulator.pdf`, 2015.

[30] Intel. What exactly is a P-state? `https://software.intel.com/en-us/blogs/2008/05/29/what-exactly-is-a-p-state-pt-1`, 2015. Retrieved on November 25, 2019.

[31] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual. `https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf`, 2016.

[32] Intel. Intel 7th and 8th Processor Family I/O for U/Y Platforms. `https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/7th-gen-core-family-mobile-u-y-processor-lines-i-o-datasheet-vol-1.pdf`, 2017.

[33] Intel. Intel Core X-Series Processor Families. `https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/6th-gen-x-series-datasheet-vol-1.pdf`, 2018.

[34] Intel. Intel® 64 and ia-32 architectures software developer's manual.

[35] Intel. IACA. `https://software.intel.com/en-us/articles/intel-architecture-code-analyzer`, 2019.

[36] Intel. INTEL® CORE^{TM} i9-9900K PROCESSOR. `https://www.intel.com/content/www/us/en/products/processors/core/i9-processors/i9-9900k.html`, 2019. Retrieved on Dezember 12, 2019.

[37] John M., B. O. Intel® Software Guard Extensions. `https://software.intel.com/en-us/articles/intel-software-guard-extensions-tutorial-part-1-foundation`, 2016. Retrieved on Dezember 10, 2019.

[38] Karaklajic, D., Schmidt, J.-M., and Verbauwhede, I. Hardware designer's guide to fault attacks. *IEEE Trans. Very Large Scale Integr. Syst. 21*, 12 (Dec. 2013), 2295–2306.

[39] Katz, J., and Lindell, Y. *Introduction to Modern Cryptography (Chapman Hall/Crc Cryptography and Network Security Series)*. Chapman Hall/CRC, 2007.

[40] Kenjar, Z., Frassetto, T., Gens, D., Franz, M., and Sadeghi, A. V0LTpwn: Attacking x86 processor

integrity from software. In (2020).

[41] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA '14, IEEE Press, pp. 361–372.

[42] Koppe, P., Kollenda, B., Fyrbiak, M., Kison, C., Gawlik, R., Paar, C., and Holz, T. Reverse engineering x86 processor microcode. In *Proceedings of the 26th USENIX Conference on Security Symposium* (USA, 2017), SEC'17, USENIX Association, p. 1163–1180.

[43] Kwong, A., Genkin, D., Gruss, D., and Yarom, Y. Rambleed: Reading bits in memory without accessing them. In *41st IEEE Symposium on Security and Privacy (S&P)* (2020).

[44] Küçük, K. A., Paverd, A., Martin, A., Asokan, N., Simpson, A., and Ankele, R. Exploring the use of intel sgx for secure many-party applications. pp. 1–6.

[45] Linux(TM). Linux CPUFreq Governors. `https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt`, 2019. Retrieved on November 26, 2019.

[46] Lynn, B. The Chinese Remainder Theorem. `https://crypto.stanford.edu/pbc/notes/numbertheory/crt.html`. Retrieved on Dezember 8, 2019.

[47] Management, D. RSA Algorithm. `https://www.di-mgt.com.au/rsa_alg.html`, 2018. Retrieved on Dezember 8, 2019.

[48] Management, D. Using the CRT with RSA. `https://www.di-mgt.com.au/crt_rsa.html`, 2019. Retrieved on November 12, 2019.

[49] Mason, P. A. Memory Basics. `https://www.egr.msu.edu/classes/ece410/mason/files/Ch13.pdf`, 2019. Retrieved on December 13, 2019.

[50] Menezes, A. J., Vanstone, S. A., and Oorschot, P. C. V. *Handbook of Applied Cryptography*, 1st ed. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[51] Murdock, K., Oswald, D., Garcia, F. D., Bulck, J. V., Gruss, D., and Piessens, F. Plundervolt: Software-based fault injection attacks against intel sgx. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)* (San Francisco, CA, USA, May 2020), IEEE.

[52] Paar, C., and Pelzl, J. *Introduction to Public-Key Cryptography.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 149–171.

[53] Panda, P. R., Silpa, B. V. N., Shrivastava, A., and Gummidipudi, K. *Power-Efficient System Design*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[54] Panwar, N., Kaur, M., and Singh, G. Performance analysis of branch prediction unit for pipelined processors. *International Journal of Computer Applications 128* (10 2015), 6–12.

[55] Patterson, D. A., and Hennessy, J. L. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[56] Payer, M. Hexpads: A platform to detect "stealth" attacks. In *Engineering Secure Software and Systems* (2016), J. Caballero, E. Bodden, and E. Athanasopoulos, Eds., Springer International Publishing.

[57] Peckol, J. K. *Embedded systems: a contemporary design tool.* Wiley, 2019.

[58] Po-Yung Chang, Hao, E., and Patt, Y. N. Target prediction for indirect jumps. In *Conference Proceedings. The 24th Annual International Symposium on Computer Architecture* (June 1997), pp. 274–283.

[59] Poledna, S. *Fault-tolerant real-time systems: The problem of replica determinism*, vol. 345. Springer Science & Business Media, 2007.

[60] Prabhu, G. M. Computer Architecture Tutorial. `http://web.cs.iastate.edu/~prabhu/Tutorial/title.html`, 2019.

[61] Qiu, P., Wang, D., Lyu, Y., and Qu, G. Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2019), CCS '19, Association for Computing Machinery,

p. 195–209.

[62] Rivest, R. L., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21*, 2 (Feb. 1978), 120–126.

[63] Rotem, E., Naveh, A., Ananthakrishnan, A., Weissmann, E., and Rajwan, D. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro - MICRO 32* (03 2012), 20–27.

[64] Rouse, M. Asymmetric Cryptography. `https://searchsecurity.techtarget.com/definition/asymmetric-cryptography`, 2015. Retrieved on Dezember 7, 2019.

[65] Schindler, W. Exclusive exponent blinding may not suffice to prevent timing attacks on rsa. pp. 229–247.

[66] Schwarz, M., Weiser, S., and Gruss, D. Practical enclave malware with intel SGX. *CoRR abs/1902.03256* (2019).

[67] Schwarz, M., Weiser, S., Gruss, D., Maurice, C., and Mangard, S. Malware guard extension: Using SGX to conceal cache attacks. *CoRR abs/1702.08719* (2017).

[68] Sundriyal, V., Keipert, K., Sosonkina, M., and Gordon, M. S. Effect of frequency scaling granularity on energy-saving strategies. *The International Journal of High Performance Computing Applications 33*, 4 (2019), 590–601.

[69] Tang, A., Sethumadhavan, S., and Stolfo, S. CLKSCREW: Exposing the perils of security-oblivious energy management. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017), USENIX Association, pp. 1057–1074.

[70] TU-Vienna. 6T SRAM Cell. `http://www.iue.tuwien.ac.at/phd/entner/node34.html`, 2019. Retrieved on November 19, 2019.

[71] Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium* (August 2018), USENIX Association.

[72] Vigilant, D. Rsa with crt: A new cost-effective solution to thwart fault attacks. In *Cryptographic Hardware and Embedded Systems – CHES 2008* (Berlin, Heidelberg, 2008), E. Oswald and P. Rohatgi, Eds., Springer Berlin Heidelberg, pp. 130–145.

[73] WCCFTECH. Intel's 6th Gen Skylake Unwrapped – CPU Microarchitecture, Gen9 Graphics Core and Speed Shift Hardware P-State. `https://wccftech.com/idf15-intel-skylake-analysis-cpu-gpu-microarchitecture-ddr4-memory-impact/4/`, 2019. Retrieved on November 26, 2019.

[74] Wikichip. Frequency Behavior - Intel. `https://en.wikichip.org/wiki/intel/frequency_behavior`, 2019.

[75] Wikichip. Kaby Lake. `https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake`, 2019. Retrieved on November 14, 2019.

[76] Wolf, M. Chapter 2 - cpus. In *High-Performance Embedded Computing (Second Edition)*, M. Wolf, Ed., second edition ed. Morgan Kaufmann, Boston, 2014, pp. 59 – 138.

[77] Zhang, T., Zhang, Y., and Lee, R. Cloudradar: A real-time side-channel attack detection system in clouds. vol. 9854, pp. 118–140.

[78] Çetin Kaya Koc. Analysis of sliding window techniques for exponentiation. *Computers and Mathematics with Applications 30* (1995), 17–24.