Andreas Kogler

Software-based Power Side-Channel Attacks

Master's Thesis

Graz University of Technology

Institute for Applied Information Processing and Communications

Advisor: Michael Schwarz
Assessor: Daniel Gruss

Graz, July 2020

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

Classical power analysis attacks have been known for decades. These attacks relied on physical access to the device under observation. Modern CPUs provide the Intel Running Average Power Limit (RAPL) software energy measurement interface, which is indented to monitor the energy consumption of CPU parts. Nevertheless, the interface can also be misused to target cryptographic primitives like RSA in classical power analysis attacks. Mantel et al. and Fusi et al. used the interface to either distinguish RSA keys or reconstruct parts of one RSA key. The general limitation of these attacks is the timing resolution of the RAPL interface.

Due to the limited capabilities of an attacker in the classical userspace threat model, the reduced timing resolution prevented power analysis attacks with higher granularity. This changed when Intel introduced Intel Software Guard Extension (SGX) to provide a trusted environment, which is shielded from malicious operating systems and other system-level software to protect the code from privileged attacks. With higher privileges, an attacker can mount precise attacks as Van Bulck et al. show, by to either advancing the execution by a single instruction or replaying the current instruction.

In this master thesis, we propose a novel approach to increase the resolution of the RAPL interface to instruction-level granularity by combining the interface with a precise execution control framework. We also extend the energy measurements from the RAPL interface by a voltage-based side channel purely in software. We show the capabilities of our approach by leaking a 71-bit long key of a 512-bit mbedTLS RSA decryption in under five minutes. The algorithm and the key are located within an SGX enclave. With the help of these power side channels, we can even leak detailed information about single instructions and use this information to reconstruct secret information in cryptographic algorithms.

We discuss why the presence of an unprivileged software-based energy measurement facility within the context of security-relevant programming is not desirable and propose countermeasures on how to mitigate these software-based power side channels.

The content of this thesis was a major contribution to a USENIX Security 2021 conference paper.

Kurzfassung

Seitenkanalangriffe über den Stromverbrauch sind schon seit Jahrzenten bekannt. Bis jetzt musste der Angreifer normalerweise Zugang zu dem Gerät haben, um die Attacken auszuführen. Moderne CPUs bieten die Intel Running Average Power Limit (RAPL) Schnittstelle an, um den Energieverbrauch von gewissen Komponenten innerhalb der CPU zu überwachen. Diese Schnittstelle kann jedoch auch missbraucht werden, um kryptographische Algorithmen, wie RSA, über den Stromverbrauchsseitenkanal anzugreifen. Mantel et al. und Fusi et al. haben die Schnittstelle verwendet, um RSA Schlüssel zu unterscheiden oder Teile von ihnen zu rekonstruieren. Allerdings waren diese Angriffe durch die niedrige zeitliche Auflösung dieser Schnittstelle limitiert.

Durch die verringerten Möglichkeiten des klassischen Userspace Angreifermodells, verhinderte die niedrige zeitliche Auflösung der Schnittstelle, präzisiere Angriffe. Das änderte sich, als Intel die Intel Software Guard Extension (SGX) einführte, um eine sichere Umgebung zu schaffen, selbst wenn das Betriebssystem bösartig ist. Van Bulck et al. zeigte, dass ein Angreifer mit diesen erhöhten Rechten im Stande ist, Instruktionen innerhalb einer SGX Enklave zu wiederholen bzw. gezielt zu einer Instruktion zu springen.

In dieser Materarbeit zeigen wir einen neuen Ansatz, um die Auflösung der RAPL Schnittstelle auf die Genauigkeit von einzelnen CPU Instruktionen zu erweitern. Wir erreichen dies durch die Kombination der RAPL Schnittstelle mit einem präzisen Ausführungs-Framework. Darüber hinaus ergänzen wir die Seitenkanalangriffe auf den Stromverbrauch mit einen spannungsbasierten Software-Angriff. Wir zeigen die Möglichkeiten unseres Ansatzes dadurch, dass wir einen 71-Bit langen Schlüssel einer 512-Bit mbedTLS RSA Verschlüsselung innerhalb von fünf Minuten rekonstruieren.

Wir disktuieren, warum die Idee einer frei verfügbaren Schnittstelle zur Stromverbrauchsmessung im Zusammenhang mit sicherheitsrelvanten Systemen keine gute Idee ist und schlagen Lösungen vor, um software-basiere Seitenkanäle auf den Stromverbrauch zu verhindern.

Der Inhalt dieser Masterarbeit war ein wesentlicher Bestandteil einer USENIX Security 2021 Konferenz Publikation.

Acknowledgements

First, I am deeply grateful to Michael Schwarz and Daniel Gruss for their support and the discussions about this thesis.

Second, I want to thank my parents Christine Sarlay and Ernst Meralla, for empowering me to focus my studies on technical topics and supporting me throughout my life.

Third, I would like to express my gratitude to Moritz Lipp for his support and the chance of being a part of a scientific publication.

Fourth, I want to thank David Oswald and Jo Van Bulck for their support and for their time to answer related questions.

Finally, I am deeply grateful to Dietrich Sullmann for supporting this thesis with equipment and giving me the freedom to fulfill my studies besides my job.

Andreas Kogler

Contents

Chapter 1

Introduction

Power analysis attacks have been known for decades [1–3]. These attacks exploit the physical energy behavior of the Complementary Metal-Oxide-Semiconductor (CMOS) technology inside CPUs to leak secret information. To mount these types of attacks, the attacker usually needs physical access to deploy the measurement equipment [4].

Modern CPUs provide the Intel Running Average Power Limit (RAPL) [5] energy measurement facility to enable energy readings without the need for additional measurement equipment. The purpose of the facility is to provide energy information for either power-aware applications or to analyze the energy consumption of large scale software. Hähnel et al. [6] showed how to use the RAPL interface to measure the energy consumption of functions, by synchronizing the measurements with the rather slow update intervals of the interface.

The possibility of measuring energy consumption via software resulted in security research on software power side channels. Mantel et al. [7] used the RAPL interface to distinguish different RSA keys based on the Hamming weight of the key. Fusi et al. [8] used the interface to reconstruct parts of an RSA key. The general limitation here is the timing resolution of the RAPL interface.

As user-space attackers have only limited control over other applications, there was no further research in this direction. However, with the introduction of Intel Software Guard Extension (SGX) [9], the threat model changed dramatically, allowing privileged attackers.

SGX was introduced with the Skylake CPU generation, to enhance the security of server-based computations. SGX is an instruction set extension aimed to provide a trusted environment even if the operating system of the computer is compromised. Since these secure enclaves must be protected from any malicious operating system, attackers have more capabilities in the SGX threat model. They can use all features of a CPU against the enclave [10].

Xu et al. [11] showed that enclaves are vulnerable to access pattern attacks because the untrusted operating system still controls the memory pages. Van Bulck et al. [12] introduced *SgxStep*, a framework exploiting timer interrupts to precisely control the execution of instructions inside the enclave. This effect can be combined with different attacks to extract side-channel information [12–15]. Murdock et al. [14] used the precise execution control framework to target specific instructions inside the enclave and apply undervolting to induce faults in a RSA computation to leak the key.

In this master thesis, we introduce a technique to record software-based power side-channel information with instruction-level granularity. We combine the RAPL interface with features from the *SgxStep* framework, allowing us to target any instruction inside an enclave. We show that we can distinguish different instructions and even distinguish branches within a single cache line. We abuse the `hlt` instruction to extend the measurement resolution in a classical Simple Power Analysis (SPA) attack. Furthermore, we discover and explore a voltage based power side channel. We achieve this by deploying an untrusted kernel module where we modified the local APIC timer interrupt.

We also show that these new techniques can be used to attack real-world cryptographic algorithms protected by SGX and reconstruct a 71-bit long key of a 512-bit ARM Mbed-TLS (mbedTLS) RSA decryption within 5 minutes. We discuss how to extend these techniques to target constant-time cryptographic algorithms.

We propose countermeasures to mitigate these software-based power side-channel attacks by disabling the updates of the voltage-based side channel and excluding instructions from the RAPL interface if they are executed inside an SGX enclave. In addition, we propose to remove the unprivileged access to the RAPL interface.

In this thesis, we close the gap between hardware and software-based attacks. The content of this thesis was a major contribution to a USENIX Security 2021 conference paper [16].

1.1 Motivation

Due to the wide power specifications of computers, the complexity of the power management features inside the CPU grew to match the energy requirements [5]. Therefore, vendors also increased the energy-related monitoring capabilities of CPUs [5], to provide the operating system or applications with the possibility to adapt to the energy consumption of the machine running the software. The RAPL interface provides these energy readings over a software interface, increasing the accessibility to energy measurements and, therefore, enables potential remote power side-channel attacks.

With the enhanced security features of SGX [17] also the threat model got extended, allowing a potential adversary to use advanced system resources to target the code running inside a from SGX protected enclave. This threat model applies to untrusted execution environment providers since these providers control the software used to execute the

customer code.

The main goal of this thesis is to extract a private key of a RSA decryption from an enclave with a classical Simple Power Analysis (SPA) attack. To overcome the limited timing resolution, we exploited CPU hardware features like the Advanced Programmable Interrupt Controller (APIC) and paging and show that the resolution of the RAPL or core-voltage side channel is no longer a limiting factor and can be extended to target single instructions.

With the attacks from this thesis, we show that power side channels are no longer expensive and complicated to mount and can be performed purely from software to target secure execution environments and even extract private key information from these enclaves.

1.2 Responsible Disclosure

We followed the practice of responsible disclosure. We reported the attacks from this thesis to Intel in November 2019. Intel issued an embargo until November 2020. Intel acknowledged the vulnerability with two CVE numbers (CVE-2020-8694 and CVE-2020-8695) (see Chapter 7).

1.3 Contributions

We make the following contributions in this master thesis.

- We demonstrate that we can enhance the RAPL interface resolution to instruction-level granularity.
- We describe a voltage-based side channel running purely in software.
- We extend a precise execution control framework to record different energy relevant side-channel information from arbitrary instructions.
- We mount an attack on a mbedTLS RSA algorithm protected by SGX and leak a 71-bit key of a RSA decryption in under five minutes.

1.4 Structure of this Document

In this thesis, we discuss the preliminaries and the general background in Chapter 2. Chapter 3 describes the threat model and the assumptions we apply to the victim and the attacker's capabilities. In Chapter 4, we explain the attack primitives and the measurement framework we use in Chapter 5 to attack the RSA implementations. We propose countermeasures to mitigate the evaluated attacks in Chapter 6. Chapter 7 discusses additional ideas and future work arising from this thesis. Finally, Chapter 8 concludes the work of this thesis.

Chapter 2

Preliminaries

In this chapter, we discuss the preliminaries and acronyms needed to understand the contributions of this thesis. We start with the general description of the power consumption of a CMOS transistor-based CPU and explain how the power consumption can be monitored and managed on Intel CPUs. We then take a look at known attacks exploiting the power consumption of a device to extract secret information.

In the next part, we look at the mathematical concepts behind the RSA public key cryptographic scheme and describe the different implementation methods and how to possibly enhance the security of the implemented algorithm by inspecting Intel SGX.

Finally, we describe the general functionality of the instruction pipeline of a modern CPU and talk about how CPUs handle interrupts and how the handling of interrupts can be exploited to control the execution of instructions.

2.1 Power Consumption of a Circuit

In 1963 the Complementary Metal-Oxide-Semiconductor (CMOS) transistor technology was developed by Wanlass et al. [18] and since then, the numbers of transistors inside CPUs kept growing [19]. These transistors are mass-produced using waver technology and kept shrinking for the past decade. Weste et al. described the CMOS technology in detail [19], the following section is based on chapters one and five of the book.

Manufacturers implement most of the CPU's low-level functionality with logic functions. These logic functions are designed to take some boolean inputs and calculate a boolean output. The essential boolean function are often denoted as gates, e.g., the `and`-gate.

To implement a logic function in CMOS logic, the function is split into the positive and negative logic part.

Where the positive part implements the conditions for the function to be *true*, and the negative part implements the terms for the function to be *false*. We show a small

example in Equation 2.1.

$$\begin{aligned} Y &= (A \wedge B) \vee C \\ \neg Y &= (\neg A \vee \neg B) \wedge \neg C \end{aligned} \tag{2.1}$$

The positive part Y is then implemented using P-type Metal-Oxide-Semiconductor (PMOS) transistors to pull the output to the high-voltage level denoted as V_{DD}, and the negative part $\neg Y$ is implemented using N-type Metal-Oxide-Semiconductor (NMOS) transistors to pull the output to the low-voltage level V_{SS}. Since a boolean logic function can never be *true* and *false* at the same time, the circuit is never directly shorted. If we did not split the logic function into these two parts, the transistors would not pull the output of the logic function to a defined voltage level, and the output would be left in an undefined floating state.

Manufactures combine these logical functions into larger and larger operations like *adders* and *state machines*. However, the power consumption of such an implementation still depends on the power consumption of the CMOS transistors.

We describe the instantaneous power consumption of a CMOS circuit with Equation 2.2. This power consumption can be split into two different parts, the static power consumption, and the dynamic power consumption.

$$P(t) = P_{static}(t) + P_{dynamic}(t) \tag{2.2}$$

The most significant part of the static power consumption resembles the power consumption due to leakage currents from the junction and the gate. These leakage currents are drawn from the power supply at any time. The dynamic power consumption depends on the state transitions a CMOS circuit is performing. The dynamic power consumption can also be split into two parts, namely the switching power consumption and short circuit power consumption. The switching power is the power drawn from switching the output from the positive to the negative logic state and vice versa. It depends on the switching frequency, the supply voltage of the transistor, and the capacitance of the circuit. We show the relation between these quantities in Equation 2.3.

$$P_{switching} \approx f_{switching} \cdot C \cdot V_{DD}^2 \tag{2.3}$$

The short circuit power describes the effect if both the PMOS and NMOS parts are closed for a short duration while switching the output voltage, therefore a short circuit current flows between V_{DD} and V_{SS}.

In Equation 2.4, we relate the instantaneous power consumption to the energy a system is consuming by integrating the instantaneous power consumption over a given period.

$$W(t) = \int_0^t P(\tau) d\tau \rightarrow P(t) = \frac{\partial W(t)}{\partial t} \tag{2.4}$$

We can calculate the instantaneous power with the derivative of the energy consumed to time. That shows us that we can use energy measurements to estimate the instantaneous power consumption and therefore get a correlation to the amount of switching behavior of the CMOS transistors inside a CPU.

2.2 Power Side-Channel Analysis

Side-channel attacks focus on extracting otherwise unavailable secret information over effects like timing information, caches, and power consumption of the hardware itself. Side-channels are often used to build more sophisticated attacks as Kocher et al. [20] showed with Spectre, or Lipp et al. [21] showed with Meltdown. The Spectre exploit tricked the branch predictor into mispredicting a branch and then letting the CPU speculatively execute a memory access to an otherwise unreachable address inside a complete valid program. This memory access resulted in a changed microarchitectural state, the cached value in the cache. The attacker can observe this changed microarchitectural state with a cache side channel and, therefore, make the speculative memory access visible.

There are many other possible side-channels like the timing delay of the Advanced Vector Instruction (AVX) unit, if it was used recently in comparison to the timing delay, if it was just powered up, as used by Schwarz et al. [22] in NetSpectre. In this thesis, we focus on side-channels based on the energy behavior of the CPU, the power side-channels. Power side-channels exploit the in Section 2.1 described dynamic power consumption of the billions of CMOS transistors of a CPU.

To measure the current power consumption of a CPU, an attacker can apply a shunt resistor before the supply voltage input and measure the voltage drop over the resistor with a high-resolution oscilloscope. The current is then calculated by dividing the voltage drop of the shunt by the resistance of the shunt. If we monitor in addition to the shunt voltage also the supply voltage of the CPU, we can calculate the instantaneous power consumption by multiplying the supply voltage with the current through the shunt resistor. Since the CPU is a clocked device, the switching inside the logic functions is also synchronized to a clock source. If the timing resolution of the measurement equipment and the noise of the measurements is low enough, we can see the power consumption of the individual instructions executed. With the possibility of measuring the instantaneous power consumption over a given time while the CPU is executing security-relevant operations, we enable classical power side-channel attacks like Simple Power Analysis (SPA) and Differential Power Analysis (DPA), which Kocher et al. showed in [1, 2].

When using SPA, the attacker exploits power leakage of operations directly. Meaning that an attacker records a few traces and then 'looks' at them, to extract a secret key.

If a cryptographic algorithm uses a conditional branch on secret data to determine the operation to execute next, an attacker can mount SPA on the algorithm. If the branch introduces a distinguishable difference in power consumption, the attacker can infer the secret key bit that leads to the branch taken or not and reason about the secret information inside the condition, if no countermeasures are in place.

If the measurement traces are noisy or the secret information is not noticeable within the SPA attack, the attacker can advance the attack to use DPA. DPA attacks use more sophisticated statistical methods to exploit even small power leakage of the hidden information. The attacker needs to interact with the device and query different inputs to the algorithm and record the power traces. Afterward, the attacker can use these traces to reconstruct parts of the key. DPA has been used to attack *DES* and *AES*, as shown by Kocher et al. [2] and Golić et al. [23].

The power consumption of operations in hardware depends on the physical implementation in hardware, e.g., executing a `xor` with an operand containing all bits set to *0* might not draw as much power as performing the same operation with a register where each bit is set to *1* since the CPU needs to flip each bit in the result. This effect is used in Correlation Power Analysis (CPA) described by Brier et al. [3]. In CPA, a model describing the underlying physical behavior of the hardware is used. The most commonly known models are the Hamming weight and the Hamming distance. The Hamming weight describes the number of set bits inside a register. In contrast, the Hamming distance describes the number of bit flips needed to change the register from the current value to another value, i.e., the energy consumed to change the register from the old value to the new one.

The model is then applied to the hypothesis of the trace and gives an estimate of the consumed power. To find the right key, a correlation coefficient is used on each of the key hypotheses, and the maximum correlation coefficient represents the right key candidate. With CPA, the attacker can attack more complex algorithms if a correct model is available.

The attacker's chance of extracting essential information depends on the quality of the traces recorded. As described by Mangard et al. [4], the power consumption for a given instance can be formalized as in Equation 2.5.

$$P_{total} = P_{op} + P_{data} + P_{el,noise} + P_{const} \tag{2.5}$$

Here the electric noise component has an expected value of zero but non zero variance. The higher the variance of the electric noise, the harder it is to reconstruct the essential data. To reduce the variance of the electric noise, multiple traces can be combined by calculating the mean of the traces. These traces must be recorded under the same conditions, i.e., the data and the electrical conditions must be the same.

If the system uses multi-core architecture, the attacker also encounters noise from dif-

ferent processes running on other cores. This noise can also be reduced by calculating the mean of multiple traces, To combine an arbitrary number of traces, they need to be time-synchronized. Otherwise, the process under observation is blurred out. Still, if another process is also scheduled synchronized with the process under observation, the noise of the two different processes does not cancel out since they are now statistically dependent. In general, the attacker needs synchronization between the measurement equipment and the execution of the algorithm under observation [4].

Power side-channels are not the only physical side-channels that can be used to extract secret information from a CPU. Genkin et al. [24] showed that a simple microphone can be used to extract an RSA key due to the high pitch noise specific operations make.

2.3 Power Management

Modern CPUs are used in many domains and have to cover a broad segment of devices. Therefore, manufacturers produce CPUs for different segments like servers or desktops. However, in these different segments, the same micro-architecture is used for high-performance computers as well as low energy computers like laptops and tablets [5].

To provide the desired performance to these devices given the energy requirements, CPUs implement complex power management. The power management is capable of reducing the overall energy consumption of a CPU by adjusting parameters like the CPU frequency, the active core count, or even disable parts of the CPU. Since the performance demands of applications vary drastically between different processes, the CPU needs to be able to adjust these parameters rapidly, as newer CPUs do in a range between 1 ms-30 ms [25].

The most common parameter for the power management to adjust is the so-called Performance States (P-states) [5]. A P-state consist of a frequency multiplicator and a voltage operating point. These quantities are often referred to as dynamic frequency and voltage scaling. The voltage operating point of the pair is often ignored in the documentations since their frequency multiplier enumerates the P-states. On Intel CPUs, P-states can be requested from each core, but the actual used P-state is shared amongst all cores. If the power management reduces the frequency multiplicator, the power consumption is also reduced as apparent from Equation 2.3.

In addition to P-states, the CPU also defines so-called Power States (C-states) [5]. C-states are used to decrease the power consumption when idling and are divided into two categories, namely the core C-states and the package C-states. The difference is that core C-states only influence the power consumption of one core, e.g., disable the clocks to reduce power. In contrast, package C-states reduce the power consumption of the whole package when all cores are idling.

The limiting factor of the maximum power capabilities of a CPU is the Thermal Design Power (TDP). The TDP is the maximum power in Watt, at which the CPU still

functions within its specifications. Since the TDP considers each of the cores running at a reasonable P-state, the TDP is often not reached, and therefore the CPU still has a thermal power budget to spend.

For this reason, Intel implemented *Turbo Boost Technology 1.0*, which gives the CPU additional P-states when the TDP is not reached to enhance the performance of the CPU as described in the Intel manual [5]. This new maximum turbo P-state depends on the number of cores that are currently not in the zeroth C-state, i.e., the operational C-state. The more cores are idling, the higher the maximum requested turbo P-state can be. The maximum turbo P-state for a given core count is shown in the `TURBO_RATIO_LIMIT` Model Specific Register (MSR). Since the thermal heat exchange is a rather slow process, *Turbo Boost Technology 2.0* introduced short time boosts, allowing the CPU to exceed the TDP for a short duration. This allows the CPU to increase the P-state for a short duration until the steady-state of the TDP is reached. The CPU offers Model Specific Registers (MSRs) to configure the boost window duration and boost window peak power consumption [5].

The operating system can request a P-state by writing to the `PERF_CTL` MSR. The CPU checks then if the request is a valid request inside the power budget and reports the currently set P-state in the `PERF_STATUS` MSR. Since the P-state is shared between all cores, but each core can individually request a different P-state, the maximum of these requests is the one used for all cores. If the operating system wants to reduce the P-state below the nominal operation point, it must clear the *Enable Hardware Coordination Bit* inside the `MISC_PWR_MGMT` MSR, otherwise, this P-state transition is blocked [5].

Additionally to the hand-triggered P-state transitions, Intel CPUs offer Hardware Controlled Perfromance States (HWP) where different P-state targets and a switching policy is set. Then the CPU handles the transitions inside these parameters. They are reducing the performance overhead of software P-state transitions. Should the CPU detect that either the TDP or another specification boundary is exceeded, the CPU starts to throttle and reduces the P-state in order to reduce the power consumption.

The Linux operating system provides two individual drivers to manage P-states, the *intel_pstate* [26] and the *cpufreq* driver [27]. Both of these are based on the described power management features from above and feature different governors to implement different types of power requirements.

2.4 Power Monitoring

In Section 2.3, we described the rather complex mechanisms acting upon the performance of a CPU. The operating system needs to control parts of the power management to provide the user with energy settings and different preferences.

To control these mechanisms from an operating system perspective, the CPU needs to provide some mechanism to monitor the current performance and power state from

the operating system. The operating system has access to the current P-state and the current throttling state of the CPU, but since the CPU is complex, estimating the power consumption only with these mechanisms is infeasible. To address this problem, Intel and AMD introduced an energy measurement mechanism built into the CPU, enabling software-based energy measurements.

Applying software-based energy measurements means that we can obtain the consumed energy of a specific part of the CPU over time without the need for additional hardware measurement equipment. On Intel, this mechanism is called Intel Running Average Power Limit (RAPL) [5] and provides a set of MSRs which allows the user to measure the energy consumption of four domains, namely the *package*, the *core*, the *uncore*, and the *memory controller*.

The RAPL register accumulates the consumed energy of the specific domain. The values in the registers are in units of microjoule, and the multiplier is specified in the `RAPL_POWER_UNIT` MSR.

With this mechanism, the operating system is capable of classifying the power consumption of different processes and even domains. This enables the operating system to adapt the power management accordingly or manipulate the scheduling of such high power tasks.

We shortly describe the four RAPL domain register below.

MSR_PKG_ENERGY_STATUS register provides measurements for the whole package, including all cores, caches, and devices which are not part of the core like buses and the builtin graphics card.

MSR_PP0_ENERGY_STATUS measures the energy consumptions of all the cores, including the non shared caches.

MSR_PP1_ENERGY_STATUS is platform-specific and can refer to a device in the uncore, e.g., the internal graphics card.

MSR_DRAM_ENERGY_STATUS provides the measurements for the memory controller.

In order to measure the energy consumption of instructions, the RAPL interface registers are used as an accumulating Performance Counter. The difference between the two counter reads is used to represent the real energy consumption of the instructions executed between the reads, as shown in Equation 2.6.

$$\Delta W = W_{end} - W_{start} \tag{2.6}$$

This difference of energy can then be divided by the period of the measurement to calculate the average power consumption of the measured instructions, as shown in Equation 2.7.

$$P_{AVG} = \frac{\Delta W}{\Delta t} \tag{2.7}$$

The Linux operating system provides a userspace RAPL driver, namely *intel_rapl*, to read the described MSRs [28, 29]. The driver gives the user the needed features to read the energy counters with no privileges if the *powercap* driver is installed. AMD recently announced also a userspace energy interface integration in the Linux kernel 5.8 [30].

The difference between the implementation of RAPL on Intel and AMD is that the MSRs on Intel CPUs are shared among cores, whereas the AMD MSRs are unique per core. This allows the measurements on AMD CPUs to be more precise since the noise of the other cores is not recorded. On the other hand, due to the shared MSR on Intel CPUs, it is possible to measure the power consumption across cores.

The RAPL interface was used for either measuring energy consumption of specific code paths or for observing the energy consumption of cryptographic primitives. We shortly summarize related work about the RAPL interface.

To measure the precise energy consumption of functions, Hähnel et al. [6] measured the update intervals of the RAPL interface. They used a second-generation Intel CPU code-named *Sandy Bridge* from 2012. They measured that the update interval for the RAPL registers is around 1 ms and has a jitter of ±50,000 cycles. In addition to the update rates of RAPL, they also implemented a synchronization method that waits for the next register update with a busy loop and then corrects the energy measurement. Rotem et al. [31] described that Intel does not measure the real energy consumed by the CPU. Instead, the CPU uses a model to estimate the consumption based on the instruction stream the CPU is currently processing. Gao et al. [32] show that the *powercap* interface is still available inside containers and can, therefore, be used to leak power information from the whole server. They also propose a model for the different RAPL domains.

Mantel et al. [7] use the RAPL interface to distinguish RSA keys. The show that they can distinguish keys by their Hamming weight with only seven measurements recorded from the *package* RAPL domain with a success rate of 99%. In contrast to the key distinguishing attack, Fusi et al. [8] use the *powercap* driver to target a 16384-bit RSA key and could determine operations that were 64 *square* operations apart.

2.5 Core Voltage

As mentioned in Section 2.3, P-states do not only define a frequency multiplier but also a voltage operating point. The voltage operating point is the second part of the dynamic frequency and voltage scaling. As Wolf et al. [33] described, the rise time of a transistor changes nearly linear with the voltage. So the performance of a CMOS transistor changes only linearly, whereas the dynamic power consumption from Equation 2.3 is influenced quadratically. Therefore, the core voltage can reduce the power consumption of the CPU without reducing the performance of the transistors with the same scale.

Register	Nr	Bits	Description
MSR_RAPL_POWER_UNIT	0x606	12:8	Energy status units in $2^{ESU}\mu J$
MSR_PKG_ENERGY_STATUS	0x611	31:0	Total energy consumed
MSR_DRAM_ENERGY_STATUS	0x619	31:0	Total energy consumed
MSR_PP0_ENERGY_STATUS	0x639	31:0	Total energy consumed
MSR_PP1_ENERGY_STATUS	0x641	31:0	Total energy consumed
MSR_PERF_STATUS	0x198	15:0 47:32	Current P-state Core voltage in 2^{-13} Volts
MSR_PERF_CTL	0x199	15:0	Target P-state
MSR_MISC_PWR_MGMT	0x1aa	0:0	Enable hardware coordination

Table 2.1: A summary of power related MSRs of Intel CPUs.

If the core voltage drops below a certain margin, the transistors start to malfunction. This undervolting effect can lead to faults in the results of CPU instructions. Faults have dramatic effects on the integrity of a CPU, as recent undervolting related papers by Qiu et al. [34], Kenjar et al. [35] and Murdock et al. [14] showed. Their attacks show that by manipulating the core voltage from software, they could inject faults into specific instructions of *RSA-CRT* and *AES-NI* implementations, resulting in leaking the secret key.

In contrast to the frequency operation point, the core voltage is not entirely fixed by the P-state. In addition to the P-state, the core voltage can be manipulated on each logical core by two MSRs. CPUs since the *Sandy Bridge* generation feature a per-core register to read the current voltage of the core [5]. The value provided by the `PERF_STATUS` MSR in bits 47 to 32 represents the current core voltage in units of 2^{-13} volts. A specific core voltage offset can be set for different operation planes by using an undocumented MSR, which was reverse-engineered by Murdock et. al. [14].

In contrast to the RAPL energy measurements, the core voltage is represented as an absolute voltage at a given time point instead of an accumulating quantity. We summarize all the related power and energy consumption MSRs of Intel CPUs from Sections 2.3 to 2.5 in Table 2.1.

2.6 Cryptographic Libraries and RSA

Cryptographic algorithms enable secure communication over untrusted channels, or to store data on cloud providers safely. The Transport Layer Security (TLS) cryptographic protocol is one of the most essential cryptographic protocols used while using the modern web [36]. Cryptographic algorithms are usually designed with resistance to known attacks in mind, i.e., the AES proposal from Daemen and Rijmen [37] stated the resistance to linear and differential cryptoanalysis. Cryptographic attacks are not the only possible attacks on cryptographic algorithms.

Cryptographic side-channel analysis, as summarized by Bauer et al. [38], exploits side-channel leakage of certain operations during the execution of the algorithm. An adversary can record the side-channel information and then reconstruct the secret key.

Injecting faults into computations is another attack vector for cryptographic algorithms. As described by Yen et al. [39], simple fault mitigations can be circumvented and still leak the secret key information.

To reduce the attack space, developers often use cryptographic libraries instead of implementing the algorithms themselves. These libraries are built with additional requirements on security and side-channel robustness in mind. One of the largest open-source cryptographic libraries is the *openSSL* library [40]. It suffered under many attacks, including the *Heartbleed* attack from 2014, which still could influence us today, as summarized by Mutton et al. [41].

A more lightweight library developed for embedded systems is the ARM Mbed-TLS (mbedTLS) [42] library. It implements the TLS protocol, including many cryptographic algorithms for applications, including RSA.

RSA is an asymmetric cryptographic scheme developed in the late 80s by Rivest, Shamir, and Adleman [43]. The algorithm remains one of the most known cryptographic schemas. The communication relies on a private and public key pair. To calculate the private and public key, the algorithm demands two prime numbers p, q, a public modulus N,

$$N = p \cdot q \tag{2.8}$$

a public exponent e,

$$1 < e < \phi(N) \tag{2.9}$$

where

$$\phi(N) = (p-1) \cdot (q-1) \tag{2.10}$$

holds. The private exponent d can then be calculated by solving

$$d \cdot e \equiv 1 \ mod \ \phi(N). \tag{2.11}$$

The resulting public key is the public modulus N and the public exponent e. The private key is the private exponent d and the public modulus N.

With the public key, a message m can be encrypted into a ciphertext c by

$$c \equiv m^e \ mod \ N \tag{2.12}$$

and the ciphertext c can be decrypted into the original message m' by

$$m' \equiv c^d \ mod \ N. \tag{2.13}$$

A conventional algorithm used for the binary modulo exponentiation of RSA is the *square-and-multiply* algorithm. Listing 2.1 shows the *left-to-right* binary exponentiation algorithm, as described in HAC 14.79 [44]. In order to keep the numbers small, we apply the public modulus N after each operation.

```
1 modulo_exponentiation(x, e, N) → y:
2   y ← 1
3   for b in bits_msb_to_lsb(e):
4     if b = 1:
5       y ← y·y mod N
6       y ← x·y mod N
7     else:
8       y ← y·y mod N
```

Listing 2.1: *Square-and-multiply* algorithm used for binary modulo exponentitaion.

It is an iterative algorithm scanning the exponent in reverse for set bits. If a bit is set, the current output temporary is squared and then multiplied with the base input of the exponentiation. If the bit is not set, the temporary output variable is simply squared. Here we can see that the exponent is directly used inside the branching condition.

The mbedTLS library implements RSA with an optimized algorithm, the sliding window exponentiation algorithm, as described in HAC 14.85 [44]. The algorithm uses a fixed exponentiation window length based on the bit size of the secret key. This window obfuscates the direct key dependency of the *square-and-multiply* algorithm. Nevertheless, as shown by Liu et al. [45], attacks mounted on a window size of *one* can be extended to arbitrary window sizes. If the window size is set to *one*, the resulting algorithm is simply, the square and multiply algorithm, as shown previously. To optimize the modulo multiplications inside the *square-and-multiply* algorithm for real hardware, the mbedTLS library uses *Montgomery* modular multiplication.

The MbedTLS library also applies exponent blinding if the user provides a randomness source. Exponent blinding is a technique where a multiple of $\phi(N)$ is added to the exponent during the modulo exponentiation, as Equation 2.14 shows.

$$d_{blind} = d + b \cdot \phi(N) \tag{2.14}$$

The exponent blinding does not affect the result since Equation 2.11 still holds. However, the blinded value changes the pattern used in the *square-and-multiply* algorithm [4]. Exponent blinding was used to prevent general timing attacks on RSA, but it only increased the amount of needed traces to recover the secret key, as Schindler et al. [46] described.

In contrast to the mbedTLS library, there are also libraries available that use different approaches to not expose the key bit inside a branch condition. These are the so-called

constant-time algorithms. Intel has provided guidelines to prevent timing attacks on cryptographic algorithms [47]. The guidelines state three main points. The first point is to make the runtime of the algorithm independent of the secret information, i.e., the runtime does not leak information about parts of the key. The second and third points state that the code and data access patterns must be independent of the secret information. Following these guidelines, an algorithm is called constant time.

To convert an algorithm to a constant time algorithm, the key bit cannot be used directly in the control flow. Instead of executing different instructions depending on the key bit, the instructions are always the same, only differing in the data of the operands. But independently of the key bit, the same data is accessed. The key bit is then often used in bit operations to select one of two inputs without exposing timing differences since the instructions are still the same.

For example, bearssl (bearSSL) [48] uses a constant time conditional `memcopy` to copy the content of a temporary variable into the accumulating variable if the key bit is set. If the key bit is not set, the copy is simply discarded. For the `memcopy`, the mathematical identity in Equation 2.15 is used.

$$x \oplus (-b \wedge (x \oplus y)) = \begin{cases} \text{x} & \text{if b} = 0 \\ \text{y} & \text{if b} = 1 \end{cases} \tag{2.15}$$

In Equation 2.15 the twos complement negation is used to transform the bit b into a mask containing all zeros or ones to then select the `xor` combination or not. If we apply the constant time copy to the modulo exponentiation algorithm as bearSSL does, we can rewrite the algorithm from Listing 2.1 to the following:

```
1 modulo_exponentiation_ctime(x, e, N) → y:
2   y ← 1
3   for b in bits_msb_to_lsb(e):
4     y ← y·y mod N
5     t ← x·y mod N
6     y ← y ⊕ (-b ∧ (y ⊕ t))
```

Listing 2.2: *Square-and-multiply* constant time algorithm used for binary modulo exponentitaion.

This implementation always executes the same instruction inside the for loop in contrast to the previous implementation. The algorithms shown omit the size requirements of the integers used in the calculations. In a real implementation, the large integer implementations also must be handled accordingly.

2.7 Virtual Memory and Paging

To grant each process its own set of virtual memory, CPUs implement paging. Paging uses multiple levels of page tables to split the virtual address space and map it to physical addresses. The lower 12-bits of a virtual address are the offset into a given page, which is usually 4096 bytes large. The remaining bits are uses to index the different layers of page tables to resolve the Page Table Entrys (PTEs) [49]. When a virtual address is used, it is send to the Memory Management Unit (MMU) to resolve the address to the physical address. Since in a real-world application, a given address is usually accessed more than once, the Translation Lookaside Buffer (TLB) is responsible for caching these virtual to physical address translations [49].

The different layers of page tables do contain not only the base address of the next page table layer but also flags describing the properties of the page tables or the page. These flags are stored in the PTEs. The *accessed* flag indicates if the page was accessed since the last reset of the flag [5]. Another common flag is the No-Execute (NX) bit, which specifies that the CPU is not allowed to execute code from this page [5].

The page table structures are set up and managed by the operating system, which is also responsible for exchanging the different mapping for each process during a context switch. If a virtual address could not be resolved over the page tables, the CPU will issue a page fault. This page fault can either be handled by the operating system by mapping a specific page to the virtual address, or by terminating the program trying to access the page [49].

2.8 Intel Software Guard Extension

Intel Software Guard Extension (SGX) [9] is a hardware-based feature introduced by Intel in 2015 with the 6^{th} generation of Intel CPUs. SGX provides Trusted Execution Environment (TEE) even if the operating system running on the hardware is compromised. This idea becomes especially useful in combination with cloud services. Since the providers have complete control over the hardware and the software providing the virtualization environment, the user has to blindly trust the cloud provider that the software and hardware are not compromised. SGX focuses on establishing a potentially safe environment for executing secret code or processing secret data on these untrusted platforms.

The general concept of SGX is to split the program into an untrusted and trusted part. The untrusted part is a normal process running on the operating system and is often referred to as an application. The trusted part is called enclave and uses the SGX protection to store and compute secret data. The enclave exposes only a defined API to the untrusted application. As summarized by Aumasson et al. [50], and Adamski [51], SGX provides many useful security-relevant features. These enclaves are protected by memory encryption to ensure the protection of the enclave data. In the enclave, a few instructions are blacklisted, to reduce the attack surface from one enclave to another

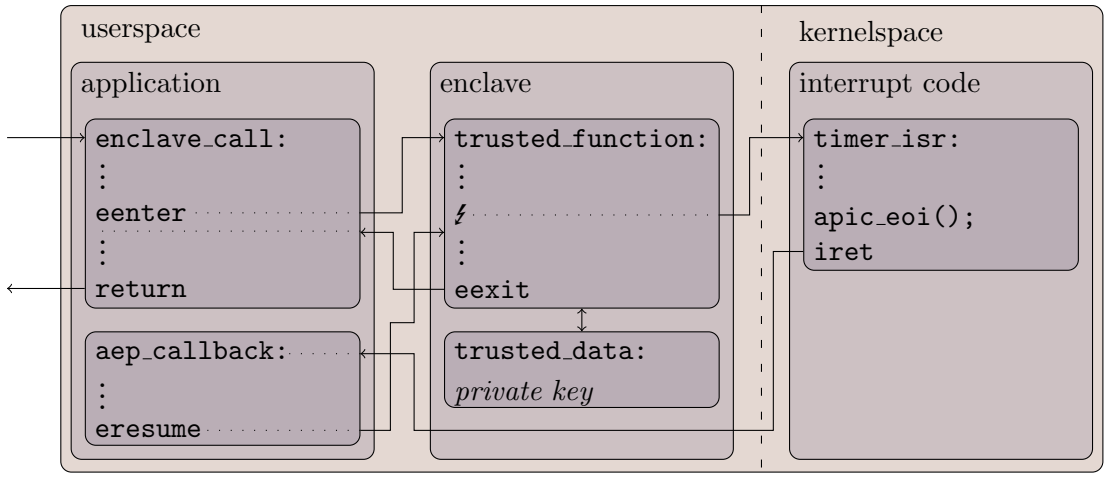

Figure 2.1: Control flow transitions between enclaves, user applications, and the operating system.

or from an enclave to the host system. The enclave cannot perform system calls, and communication with the operating system must be routed over the untrusted application. The threat model features trusted execution even if the operating system, the BIOS, the system management mode, or even the management engine is compromised as summarized by Costan et al. [52].

To support encryption, the Enclave Page Cache (EPC) was introduced, which is a unique encrypted memory region. The EPC lies in the Processor Reserved Memory (PRM) region of the CPU. This region is protected from memory reads even from the Direct Memory Access (DMA) controller. SGX sees the DRAM as an untrusted resource and, therefore, uses cryptographic primitives to store data there [52]. The Memory Encryption Engine (MEE) enforces the encryption of enclave data stored outside the EPC. The enclave defines a EnclaveLinear Address Range (ELRANGE) in the virtual memory, which contains the enclaves data and code stored on pages in the EPC. If the untrusted application tries to access a virtual address form the ELRANGE the read access returns only data with all bits set to one [52].

The MEE is described in detail by Gueron et al. [53]. The EPC is limited in size, but the pages can be swapped to the system memory in an encrypted form. Although the MEE manages the encryption of these pages, the paging mechanism is still the same as with a regular page. Therefore, the operating system has control of the paging control structures and the paging flags. It also has the responsibility to set the access rights of these pages accordingly.

An enclave can only be entered and exited with the `eenter` and `eexit` instructions. The entry and exit points are defined during compile time. To ensure normal system coexistence with other processes or threads running concurrently, SGX also supports Asynchronous Enclave Exits (AEX) to handle interrupts and faults. AEX stores the

execution context into the EPC and then gives the execution to the interrupt or fault handler [52]. After the interrupt handling is done, the handler returns to a callback in the unsafe region of the application pointed by the Asynchronous Exit Pointer (AEP). Inside the callback, the `eresume` instruction is invoked to restore the previously stored execution context from the EPC and resume the execution in the protected environment.

The enclave can disable the PMC interface by using the Anti Side-Channel Interface (ASCI) feature of the CPU. This feature changes the behavior of the PMC interface, but the Intel documentation does not provide enough information about the feature, as mentioned by Costan et al. [52]. So it might still be possible to use some PMC events to determine information about the enclave [52].

SGX enclaves can either be built in debug or release mode. In the debug mode, the enclaves memory and register can still be read with the `edbgwr` and `edbgrd` instructions. During a context switch, the enclave registers are stored in the Save State Area (SSA). With these debug instructions, the user can read the registers stored in the SSA for debugging purposes. If the enclave is built in release mode, these instructions are blocked. To build an enclave in release mode, the user must request a signing key from Intel [54].

Despite these mitigations, security researches have successfully mounted attacks on SGX. Schwarz et al. [10] summarized and categorized the recent attacks. The first category are side-channel attacks like Xu et al. [11] and Van Bulk et al. [55]. They used the paging mechanism to either unmap pages or exploit page flags to extract access patterns. The second category are the transient execution attacks, which exploit speculative execution and out-of-order execution (see Section 2.10). *Zombieload* [15] and *RIDL* [56] are capable of leaking data from internal CPU buffers. The last category uses fault-based attacks, as mentioned in Section 2.5. Murdock et al. [14] and Kenjar et al. [35] used undervolting to inject faults into an enclave to attack cryptographic algorithms.

SGX also provides additional attack space by enabling the adversary to write undetectable malware, which is capable of spying on the system as shown by Schwarz et al. [57]. They also showed that even if the `rdtsc` instruction is blacklisted inside the enclave, an attacker can still use a counting thread to create a timing source for cache timing attacks. Van Bulck et al. [12] showed that enclaves are vulnerable to precise execution control attacks, as we summarize in Section 2.11.

2.9 Advanced Programmable Interrupt Controller

Operating systems nowadays handle many different tasks or processes running seemingly concurrent. Since the number of processes and threads running on an operating system are higher than the number of hardware cores provided, these tasks have to be preemptive or cooperative. If the tasks are cooperative, they yield the control flow in specific points of their execution to other tasks. Cooperative multitasking is often used if all the task running are known, and it can be guaranteed that each task gets a fair share of the execution time [49]. For a desktop operating system where the tasks running on it are

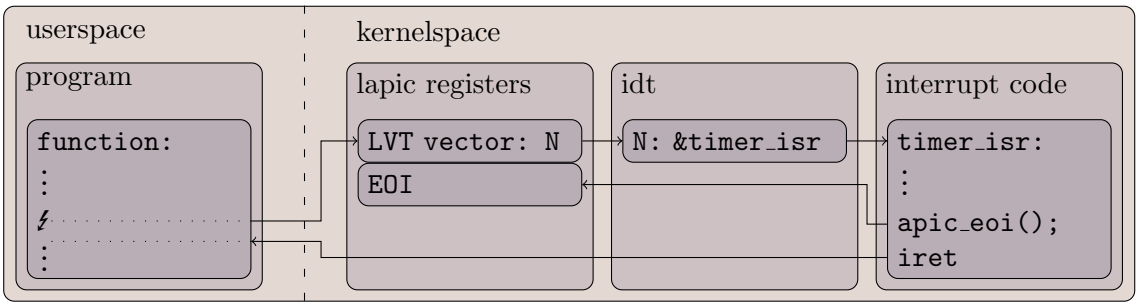

Figure 2.2: Interrupt handling of an risen local APIC interrupt.

unknown, this method is not applicable. So in order to give each task a fair share of the execution time, the control flow must be given back to the operating system at some point. However, if the task never yields their execution to the operating system, it could never preempt the task [49]. To solve this problem, a timer interrupt is used to generate an interrupt after a specific time, to interrupt the currently running task and schedule the execution back to the operating system, which then can decide to perform a context switch to execute another task or just continue with the current task.

The timer interrupt is generated by the Advanced Programmable Interrupt Controller (APIC) [5]. Since multi-core CPUs provide more then one core, each of the cores has a local APIC. The local APICs are connected over a bus interface with each other and the IO-APIC [5]. The IO-APIC handles external interrupt requests from CPU external hardware and routes the request to the local APICs.

The local APIC implements different registers for configuration and error checking. Depending on if the local APIC is running *x2apic* mode or in the *flat* mode, the registers can either be accessed by MSRs or by memory-mapped IO [5]. For local interrupt handling, the local APICs come with Local Vector Tables (LVTs). These LVTs determine which interrupts are routed to the CPU and to which interrupt vector the interrupt is wired.

The local timer interrupt features in addition to the LVT of the timer, a configuration register to let the timer run in one of three modes:

One-shot mode uses a simple counter register to count down from a set-value. As soon as the value reaches zero, the controller sends the interrupt to the configured vector.

Periodic mode is the same as the *one-shot* mode, but the value is reset to the set-value after the counter reaches zero.

TSC deadline mode uses instead of the relative timer increment of the other two modes, an absolute timestamp to generate the interrupt. The benefit of this configuration is the low jitter and the possibility to trigger interrupts concerning an absolute time.

Figure 2.2 shows the handling of a timer interrupt. After an interrupt was triggered either by the local APIC or IO-APIC, the CPU looks up the dedicated interrupt vector from the corresponding APIC LVT. With the interrupt vector, the CPU determines the Interrupt Service Routine (ISR) from the Interrupt Descriptor Table (IDT). The ISR is a function defined by the operating system or device drivers to handle the risen interrupt. After the ISR finished the interrupt handling, the APIC interrupt needs to be acknowledged. This is done by writing to the local APIC End Of Interrupt (EOI) register. The handler can then execute the `iret` instruction to give the control flow back to the previously executed code [5]. There are different types of interrupt handlers, which can be defined by the descriptor of the vector in the IDT. 'Trap' gates keep the interrupts enabled during the entry of the ISR and 'interrupt' gates disable interrupts during entry of the ISR. A particular case for interrupt handling are the Non-Maskable Interrupts (NMIs). These are special types of interrupts that cannot be disabled (masked) and can occur during disabled interrupts. NMIs are often used to detect system lockups and to implement watchdogs.

The local APIC controller has recently been used to mount so-called precise execution control attacks, where the victim is interrupted every few cycles, enabling the adversary to generate a timer interrupt after each instruction [12]. We describe these types of attacks in detail in Section 2.11.

2.10 Pipelining and Out-of-Order Execution

Modern CPUs rely heavily on pipelining to increase their execution speeds. We summarize the architecture of a pipeline and how the CPU handles out-of-order execution in this section.

The classical view of a CPU pipeline executing machine code instruction is the 'fetch', 'decode', 'execute', and 'writeback' cycle as described by Hennessy et al. [58]. The CPU cycle starts execution by fetching the current instruction from the memory address pointed by the instruction pointer. The memory access will probably be served from the instruction cache or result in an instruction cache miss if the CPU did not predict the instruction address correctly. After the instruction is loaded from the memory, it is decoded, and the register or memory operands are determined. Since the CPU can dynamically rename the registers, the register operands are looked up in the register file. After the operands are ready, the instruction is executed, and the results are written back to the output operands. If the instructions were either a store or a load, the handling is different because of the complex memory hierarchy of a CPU.

To meet the performance requirements of modern CPUs, this cycle is no longer that simple. Instead of waiting until the writeback stage is finished to issue the next fetch, the CPU immediately tries to fetch the next instruction after the previous fetch finished. So instead of waiting for each pipeline stage to finish, the CPU uses each stage continuously. In order for the fetch stage to know which instructions follow the current one branch

predictors are implemented in hardware. These predictors keep track of control flow branches and try to predict the next instruction to be executed.

To reduce the complexity of modern Complex Instruction Set Computers (CISCs), the CPU breaks the instruction defined in the machine code into finer-grained micro-instructions during the decoding phase. Instead of executing these micro-instructions in sequential order, the CPU analyses the dependencies of the operands concerning other micro-instructions and tries to minimize the stalling time by reordering them. The stalling time is the time which the CPU is idle and cannot execute micro-instructions because of dependencies among the operations itself or due to the lack of free hardware to execute the micro-instruction. The reordering is based on different aspects, e.g., the CPU can execute the micro-instruction before another because the operands are independent of each other, or the micro-instruction must wait for another micro-instruction to finish first.

After reordering, the micro-instructions are scheduled to the different execution ports of a CPU. These ports represent hardware components of the CPU where each core has a given number of ports capable of executing specific instructions. On x86 architectures, the instructions retire in order, meaning that the possible out-of-order executed micro-instruction has to wait for their predecessor to finish. This *in order* retirement is achieved with the so-called reorder buffer.

Due to mispredictions of the different predictors that predict the next instruction to execute, it can happen that an already finished micro-instruction was falsely executed. If this happened, the CPU must revert the effects of the executed micro-instructions and flush (clear) the pipeline, because it is filled with micro-instructions from the wrong instruction address. The CPU can revert the architectural state, like the internal registers and the memory content, but the state of, e.g., the caches is not reverted, and this enabled the Spectre type attacks as summarized by Canella et al. [59]. Here a differentiation between the architectural and the micro-architectural state is made, whereas the micro-architectural state can be observed by side-effects in different components of the CPU. After the mispredict is corrected, the real instruction is issued.

In addition to pipeline flushes due to mispredictions, the CPU also needs to flush the pipeline if an interrupt or fault rises. Depending on when the interrupt arrives during the execution of the instruction, it can retire or be stalled and reissued after the interrupt was handled. Since the ISR uses regular instructions to handle the interrupt, the pipeline must be flushed, and the control flow must continue at the given entry point of the ISR. We described the control flow transitions and lookup of the ISR in Section 2.9.

From the microarchitectural behavior of a CPU multiple fields of attacks arise. The *Spectre* attacks exploit control or data flow misprediction, whereas the *Meltdown* based attack exploits transient execution after a fault occurred [20, 21, 59]. Mircoarchitectural Data Sampling (MDS) attack types enable an adversary to leak data from different microarchitectural buffers [15, 56, 60].

2.11 Precise Execution Control

Precise execution control abuses CPU features to gain control over the control flow of a victim. The following frameworks all require kernel privileges in order to modify the mechanism used for the attack and are, therefore, only suitable for the SGX threat model.

The *PTEditor* by Schwarz [61] allows control-flow monitoring at page size level. An attacker can use this to monitor the execution path through the victim's control flow and possibly detect secret information due to conditional function calls that are placed on different pages.

MicroScope by Skarlatos et al. [62] uses page faults to precisely replay the transient execution of a few instructions. This replay can be used to record fine-grained side-channel information of the instructions since it can be repeated indefinitely.

SgxStep by Van Bulck et al. [12] uses the local APIC timer interrupt to interrupt the CPU during the execution of a given instruction and forcing the instruction to stall and be reissued after the handling of the interrupt is finished as we described in Section 2.10. This is achieved by using the local APIC timer in one-shot mode with a small trigger value, as we mentioned in Section 2.9. The attacker can start the timer after a known event occurred, e.g., a page fault. To trigger the page fault, the adversary must know the address of the code page containing the code to collect side-channel information from. The address can be found by either knowing the offset of the code from the enclave base address or by using, e.g., the *PTEditor* to monitor the pages accessed by the enclave after calling a specific API and determine a target page.

If the enclave is loaded from the file system, the code of the enclave is unencrypted. Aumasson et al. [17] explain a way to establish encrypted enclave code. The enclave generates a private and public key pair and then use the public key with some authentication information to receive an encrypted blob of code from a trusted source. The code is then decrypted with the private key inside the enclave. This method enables non-reverse engineerable enclaves.

If the code is known and the first page fault address is found, the adversary sets the No-Execute (NX) bit in the page table entry. This is possible since the operating system still manages the enclave pages. If the enclave tries to execute the code from the given page, the page fault is triggered. Now the adversary can clear the NX bit and return from the page fault handler.

This will then give the control flow to the AEP callback, which then invokes the `eresume` instruction. To precisely set up the next local APIC timer interrupt, the AEP callback must be overwritten. The AEP callback is located in the untrusted application, more precisely in the SGX-SDK library. *SgxStep* uses a modified version of the SDK library, which provides functionality to overwrite the default AEP callback.

The timer then triggers the interrupt after the specified clock ticks have passed. If the

interrupt arrives during the execution of the instruction and forces the CPU to stall and reissue the instruction, this is called a zero-step. If an instruction is zero-stepped, the CPU will not advance the instruction pointer. If the triggered timer interrupt arrives after the retirement of the instruction, we can either observe a single-step if the instruction pointer is advanced one instruction, or a multi-step if the instruction pointer is advanced more then one instruction.

Note that the instruction pointer of an SGX enclave is only visible in debug enclaves, as mentioned in Section 2.8. So the detection of single- and zero-steps has to be managed differently. Van Bulck et al. used the paging flags of the code page containing the enclave's code as a replacement. If the instructions retire and the CPU advances the instruction pointer, the accessed bit of the page gets set. If the instruction was interrupted by the timer interrupt and is stalled by the CPU and the complete architectural state is rewinded, the access bit of the code page is not set.

So to detect single- and zero-steps, the attacker clears the accessed flag of the code pages in the AEP callback before returning the execution to the enclave. After the timer interrupt was triggered, the attacker can then check if the code page was accessed to determine if a singe- or zero-step occurred. Multi-steps cannot be detected with this method. We denote this technique as instruction counting instead of the CPU instruction pointing.

Chapter 3

Threat Model

In this chapter, we discuss the threat model and the attacker's capabilities.

3.1 High-Level View of the Attacks

In our threat model, we assume that the victim and the attacker are running on the same physical machine. The attacker has complete control of the operating system running the victim's code. This scenario is feasible for cloud servers, where the vendor has complete control over the virtualization environment used to execute the customer's code. We assume that the victim and the attacker are running on an Intel CPU with Intel Software Guard Extension (SGX) support. The oldest generation of Intel CPUs, which also includes the RAPL interface, is the Skylake generation.

Our attack makes no assumptions on the use of secure boot as the root of trust on the target platform. Enclaves cannot verify the root of trust because the input operations have to go through the untrusted operating system. SGX enclaves only verify the integrity of the CPU they are running on and not the operating system nor the root of trust.

Our threat model adopts the SGX [50] threat model, where the enclave should be protected from an untrusted or malicious operating system. In our attack, we do not exploit bugs in the enclaves source code nor the source code used for loading enclaves.

3.2 Victim

The victim can run an unprivileged program inside an SGX enclave, excluding the instructions blacklisted inside an enclave (see Section 2.8). The victim provides a API to trigger a cryptographic RSA signing process, and the victim does not apply limitations on the number of times the attacker can trigger the component. We assume two different models for the victim algorithm. In the first model, we do not need any information

about neither the plaintext nor the keys used inside the algorithm. In the second model, we assume that we have a known-plaintext attack, i.e., we know the currently used plaintext of the victim. We require the second model to reconstruct intermediates of the RSA algorithm to circumvent the plaintext-dependent instruction offsets to mount one of our attacks.

The keys for the cryptographic component are kept in the secure enclave memory and are not accessible for the attacker directly. The enclave also executes the same instructions if the API is triggered multiple times. The victim does not exchange encrypted source code with a trusted domain, and therefore, the source code is available to the attacker. However, the attacker cannot use the key from the source code directly. We do not exploit any bugs inside the implementation of the cryptographic algorithm nor the code used to provide the API.

3.3 Attacker

The attacker has access to all CPU features and the whole operating system and can execute privileged code. The attacker uses the API provided by the victim to issue a signing process and can trigger unlimited invocations of the cryptographic algorithm and also knows the plaintext currently used in the enclave if not mentioned otherwise. The attacker can record multiple traces of the same invocation of the victim API. The target of the attacker is the modulo exponentiation algorithm (see Section 2.6) inside the enclave. The victim uses different types of RSA implementations, but the key is only reconstructed by targeting the algorithm. The attacker has access to the source code of the enclave if not stated otherwise.

Chapter 4

Attack Primitives

In this section, we introduce the three basic mechanisms for our attacks. First, a *measurement framework*, which exploits that RAPL can be used as a power side channel, even if the code is protected by SGX. We use this *measurement framework* to attack RSA implementations inside SGX in Chapter 5. Second, we exploit the local APIC to enable *zero*- and *single*-stepping in Section 4.4. The *zero*- and *single*-stepping technique provides instruction-level power side-channel resolution for our attacks on SGX, similar to *SgxStep* [12], *Nemesis* [13], and *MicroScope* [62]. Third, we extend the *measurement framework* to precisely hit target instruction and describe how to determine target instruction in Sections 4.5 and 4.7.

4.1 RAPL as a Power Side Channel

To mount a power side-channel attack, the attacker usually needs physical access to the device or CPU to deploy the measurement equipment and measure the power consumption of the device while running the victim's code in a synchronized way. Physical access to a device limits the attack space for remote attacks but is still considerable for untrustworthy cloud providers since they have access to the servers.

In this section, we explain how we use the power monitoring features from Section 2.4, to construct a software power side channel. We exploit the general idea that instruction or even operands have a unique power consumption, and the RAPL software interface provides the power consumption over a given time window. We can use the software interface to generate energy readings without additional measurement hardware and, therefore, lift the restriction of direct physical access to the device. We also reduce the effort to implement a synchronization technique between the interface and the function we want to measure.

The increment in the different RAPL domains can leak information about different instructions executed with a specific granularity. If we compare it to cache side-channels

where the timing difference of a memory access leaks only the level from where the memory access was served, we can gain more information over a power side channel than a cache side channel.

We can read the RAPL interface over either the dedicated *intel_rapl* domain files or directly by using the `rdmsr` instruction. To determine the performance of the power side channel, we first look at the update intervals of the register, i.e., we detect the time it takes for the RAPL interface to change to a newer measurement value. The update interval corresponds to the maximum timing resolution of our side channel. We repeat the measurement for the update intervals of the RAPL interface to validate the intervals found by Hähnel et al. [6], which were around 1 ms, with a jitter of ±50,000 cycles.

We first look at the update intervals of the `rdmsr` instruction. To measure the update rate, we busy-wait until the MSR receives a new value and then start measuring the time until the next new value arrives. We do this 20,000 times and plot the timing differences as histograms. During the measurement of the update intervals, we use the *stress* program on the other cores, to make sure that the system is consuming energy and does not fall into a lower C-state, i.e., that the energy consumption is above the increment of the RAPL interface defined in the `RAPL_POWER_UNIT` MSR.

We show the code used to record the Time Stamp Counter (TSC) differences in Listing 4.1. Since the `rdmsr` instruction needs *ring 0* access, we run the code inside a Linux kernel module.

```
1 uint64_t measure_msr_timing(uint32_t number, uint64_t mask) {
2     uint64_t new, last = rdmsr(number) & mask;
3     while( last == (new = rdmsr(number) & mask) );
4     last = new;
5     uint64_t start = rdtsc();
6     while ( last == (rdmsr(number) & mask) );
7     uint64_t end = rdtsc();
8     return end - start;
9 }
```

Listing 4.1: Measuring the update interval of a specific MSR.

Table 4.1 and Figure 4.1 show the results of the timer interval measurements for an *Intel(R) Xeon(R) CPU E3-1275 v5 @ 3.60GHz* CPU. In the table, the argmax fields respond to the most common update interval.

We see a different update interval for the *core* domain in comparison to the *package* and *dram* domain. Instead of the documented update intervals of 1 ms by Intel [5], the *core* domain update interval is nearly twenty times faster as the *package* and *dram* domains. The *core* and *dram* RAPL domain shows an update interval of around 0.988 ms. From the histogram, it is visible that the *package* and *dram* domain also have a second cluster centered around 1.03 ms. The *core* has a faster update rate then documented, with only 43.3 µs. In addition to the RAPL domains, we also measured the core-voltage update interval, which is 74.8 µs. In the histogram of the core-voltage domain, we can also see

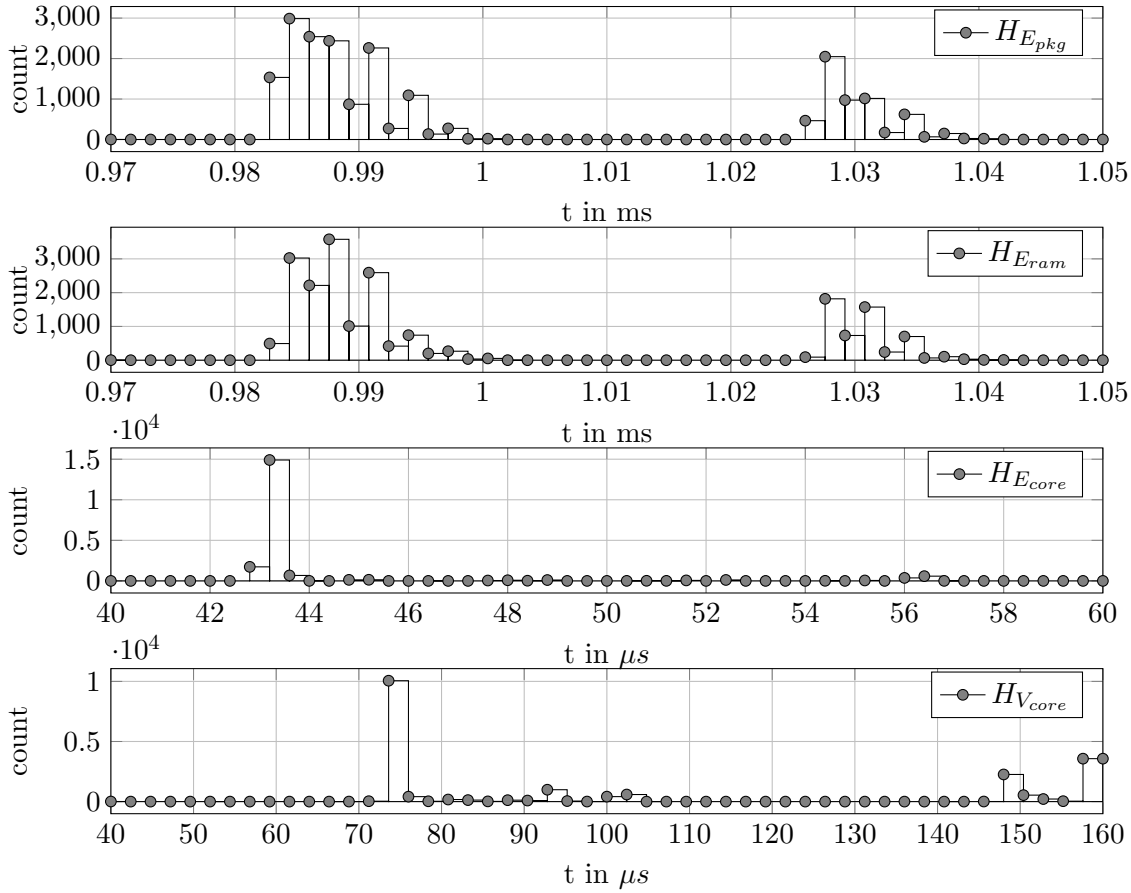

Figure 4.1: Timing histograms of the different RAPL and core-voltage MSRs.

harmonics on each multiple of the above update interval. The reason for the harmonics is the different register type. RAPL uses an accumulating register, and therefore, the values are monotonically increasing, whereas the core-voltage register returns the instantaneous voltage value. So if the voltage value gets updated with the same value, we cannot detect the update with our measurement algorithm. From the percentiles of Table 4.1, we can see that the *package* and *dram* domain show fewer outliers than the *core* domain.

When using the Linux userspace *powercap* RAPL driver, we can also observe update timings around the ones when using the `rdmsr` instruction directly with some minor overhead for the *core* domain of around 8 µs. This enables potential attacks from an unprivileged application on the kernel or an SGX enclave. We shortly discuss the possibilities in Chapter 7. In this thesis, we read the RAPL interface and the core voltage in a privileged context and directly over the `rdmsr` instruction to reduce the overhead and have access to the core-voltage register.

Domain	Argmax	Mean	Median	90^{th} Percentile	95^{th} Percentile
package	0.987 ms	1.000 ms	0.991 ms	1.031 ms	1.033 ms
dram	0.988 ms	1.000 ms	0.991 ms	1.031 ms	1.034 ms
core	0.043 ms	0.044 ms	0.043 ms	0.044 ms	0.056 ms
core voltage	0.075 ms	0.117 ms	0.076 ms	0.224 ms	0.259 ms

Table 4.1: Update times of the registers.

4.2 Measurement Framework

We take advantage of the short update intervals when reading the RAPL domains directly via the `rdmsr` instruction. For that purpose, we build a Linux kernel module, so we can execute our measurement routines in *ring-0* to use the privileged instruction. On top of the kernel module, we build a library for interacting and configuring the module. We denote these two components as *measurement framework*. The *measurement framework* is then used to collect power side-channel traces. We denote the collection of traces with the *measurement framework* as *measurement process*.

As we describe in Section 2.4, the RAPL interface MSRs are accumulating counters. Therefore, the measurement has to be split into *start*- and *stop*-measurement. Hähnel et al. [6] used a synchronization technique to wait for the next update of the RAPL register. Due to the slow update intervals of the interface, it can happen that the value of the RAPL domains did not change between the *start*- and *stop*-measurement, even though the CPU consumed energy. Therefore, a synchronization technique is needed to wait for a new register value.

We implement a similar technique, but instead of synchronization before each read of the RAPL registers, we only synchronize our measurements in the *start*-measurement routine, and we do not account for the energy consumed during the synchronization. We discuss our `hlt` instruction delay technique to handle samples where we did not observe an update at the end of this section.

The synchronization is done by waiting for one of the RAPL domain registers to change. In our implementation, we wait for the *package* domain to change, because this will also give the other registers enough time to update. A caveat here is that the updates of the different RAPL registers might not be synchronized to the slower *package* domain. In practice, we did not observe a negative effect of not synchronizing each register. Due to the synchronization, we get an overhead of 1 ms maximum per sample recorded.

We calculate our measurement sample from the *start*- and *stop*-measurement after the data for the *stop*-measurement is recorded. Listings 4.2 and 4.3 shows the source code for the *start*- end *stop*-measurement. Our measurement sample includes seven fields, whereas five of them contain power-related information, as we describe below.

TSC Difference This field contains the Time Stamp Counter difference between the *start*- and *stop*-measurements. This field can be used to calculate the average power consumption, as we described in Equation 2.7.

RAPL Differences These fields contain the energy differences between the *start*- and *stop*-measurements for the *package*, *core*, and *dram* RAPL domain.

Core Voltage This field contains the core-voltage operation point read from the MSR during the *stop*-measurement. We decided only to read this register once because the register itself is not accumulating. It also would not make sense to read it in the *start*-measurement, because the value would not contain information about the target instructions.

P-state This field contains the currently active P-state during the *stop* measurement. The current P-state is read from the same MSR as the core voltage, and we can read the value with no overhead. As we describe in Section 4.8, the P-state is essential for monitoring the *measurement process*.

SGX Debug This field is a monitoring field only used to configure our attacks. We use it to store a specific register from the Save State Area (SSA) [50] from an enclave to get additional information for each sample. This becomes especially useful to determine the position of instructions, as we describe in Section 4.7.

We store each of the fields as a 64-bit unsigned integer inside a packed structure. These samples are then stored in a buffer within a control structure allocated with `kmalloc`. We memory map the control structure, including the buffer into the userspace, so the library can access the buffer without additional overhead and can modify the configuration of the module.

```
1  struct measurement_t start;
2
3  void start_measurement() {
4      uint64_t last = rdmsr(PKG);
5      while (last == rdmsr(PKG));
6      start.tsc = rdtsc();
7      start.pkg = rdmsr(PKG);
8      start.pp0 = rdmsr(PP0);
9      start.ram = rdmsr(RAM);
10     execute_hlt_delay();
11 }
```

Listing 4.2: *Start*-measurement function.

```
1  struct measurement_t stop;
2
3  void stop_measurement() {
4      stop.tsc = rdtsc();
5      stop.pkg = rdmsr(PKG);
6      stop.pp0 = rdmsr(PP0);
7      stop.ram = rdmsr(RAM);
8      stop.sts = rdmsr(STS);
9      stop.dbg = rdssa();
10     write_sample(&start, &stop);
11 }
```

Listing 4.3: *Stop*-measurement function.

To record a measurement sample, we first introduce two new terms. The trigger event specifies an event, after which a sample should be recorded. The trigger type describes the behavior of the measurement module after a trigger event occurred, i.e., how the measurement sample is recorded. We implement the following two trigger types.

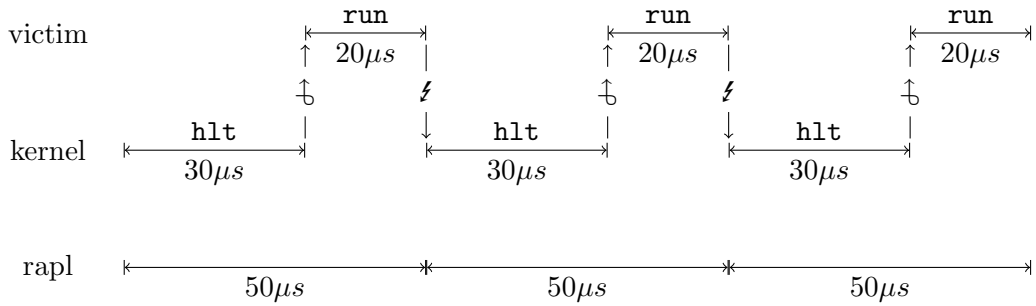

Figure 4.2: Timing squence diagram of the victim and the modified timer interrupt with a `hlt` delay of 30 µs.

***Symmetric* Trigger** When using this trigger type, every time a trigger event occurs, the kernel module will first perform a *stop*-measurement and then perform a *start*-measurement. This trigger type will never stop measuring and is therefore useful for a continuous stream of samples. Note that the first measurement of this trigger type must be discarded since the first sample never received a *start*-measurement.

***Asymmetric* Trigger** This trigger type will keep track of the number of trigger events happened and perform a *start*-measurement for each odd event, and the finalizing *stop*-measurement for each even event. This trigger type is useful when we have an explicit start and stop point in our observation.

We can use these trigger types with trigger events to record side-channel information. The most straightforward trigger event we implement is an `ioctl` system call to our kernel module to send a trigger event. This trigger event is sufficient to measure procedures with a clear, distinguishable energy signature. The `ioctl` system-call has a significant overhead and is therefore not suitable for a periodic high-frequency sample stream.

Intel's RAPL implementation measures the energy consumption of a specific domain over the whole package (see Section 2.4). This means that, e.g., the *core* domain includes the energy consumption of all cores, including all non-shared caches. We can use this property to dedicate one core for our energy measurements and still observe the energy consumption of all cores. This works only for the RAPL registers, the core-voltage MSR is unique per core and, therefore, can only be read from the given core.

We decided against the monitoring core concept since we lose the core-voltage information. Instead, we reduce the noise of other cores as much as possible by disabling all cores except two. From these two cores, we use one as a general-purpose core to keep processes running, and isolated the other. CPU cores can be disabled by using the Linux kernel parameter `max_cpus` and isolated by using the `isol_cpus` parameter list. The isolated cores are removed from the scheduling list.

We denote the isolated core as the victim core. We manually schedule the measurement kernel module and the program under observation to the victim core, by using the *pthread*

set core affinity mask function `pthread_setaffinity_np`. We can also achieve this by using the *taskset* program.

We want the possibility to record periodic samples, like a classical attacker using hardware power side-channels has (see Section 2.2). To do so, we take advantage of the local APIC. The APIC timer interrupt can be configured in the so-called periodic mode. In this mode, the local APIC generates timer interrupts with a fixed interval. Alternatively, we can also implement periodic sampling by using a kernel thread or kernel timer. However, if we use the kernel thread, we still need a busy loop to check for the timestamps when to trigger the measurements, and this introduces an energy overhead. Since we run our kernel module and the process under observation on the victim core, we also have a scheduling problem. If we use a kernel timer, we simply add overhead to the local APIC timer interrupt, since the timer interrupt triggers the kernel timers. To sample periodically, the local APIC timer interrupt must be configured in periodic mode, with the *start*-measurement synchronization disabled or either in one-shot or TSC deadline mode where we set the next timer interrupt trigger value to a fixed interval.

To use the timer interrupt as a trigger event for our measurements, we need to hijack the timer interrupt handler. We use the exported kernel *kallsyms* symbol to look up the local APIC structure from the victim core and modify it. We store the current handler so we can restore it after we finished the *measurement process*. The handler can now be overwritten with our custom interrupt handler, which provides the trigger event.

We do not implement a synchronization to the RAPL registers in the *stop*-measurement function. Instead, we implement a delay function. The delay function is used to block the victim's execution until the RAPL register receives an update. To implement a timed delay inside the local timer interrupt, we decided to use the `hlt` instruction. The `hlt` instruction stops the execution of the core until an external interrupt arrives. Using the instruction directly can completely lock up the CPU if we execute it with interrupts disabled. The Linux kernel already provides a safe `hlt` function, which enables the interrupts before calling the instruction.

Calling the instruction stops the core from executing instructions. Therefore, the energy accumulated by the different RAPL domains should be at a minimum in comparison to other delay techniques. We observed a reduction in the power consumption by factor 2.06 for the *core* domain in comparison with a `nop` delay loop with a fixed CPU frequency of 90% of the maximum frequency.

Figure 4.2 shows the general principle of the `hlt` delay. To continue the execution of our *start*-measurement function, we need to generate an interrupt. We can simply use the timer interrupt in one-shot mode and trigger an additional timer interrupt. We must handle this timer interrupt differently than the trigger event interrupts to not trigger an additional *start*-measurement.

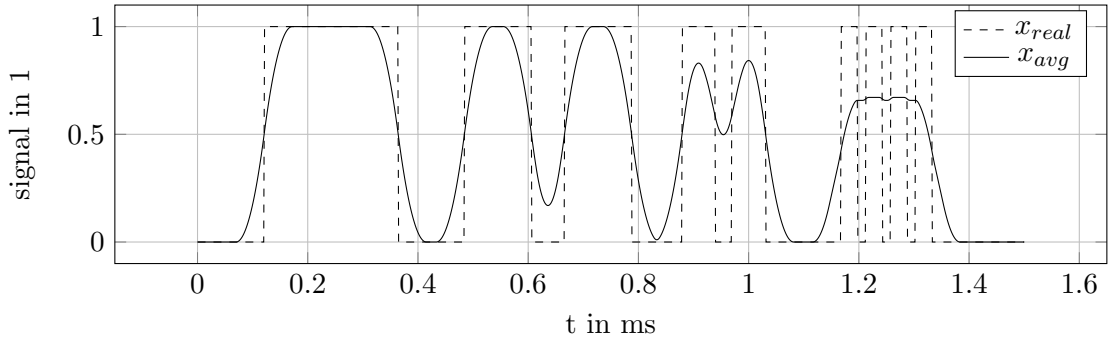

Figure 4.3: Loss of information due to the filtering process.

4.3 Simple Power Analysis with RAPL

In this section, we focus on reconstructing power side-channel information with the help of the *measurement framework* and explore the SPA approach (see Section 2.2).

The Running Average Power Limit interface is a running average filter, as the name suggests. Therefore, the values represent filtered values instead of the real energy consumption. Since we know the update intervals of the registers from Section 4.1, we can simulate the theoretical limits of the *measurement process* due to filtering. Running or moving average filters can be represented as a convolution of the original signal and a filter kernel, as we show in Equation 4.1.

$$x_{avg}(t) = x_{real}(t) * k_{avg}(t) \tag{4.1}$$

Equation 4.2 shows the filter kernel of a moving average filter, which is simply a rectangular window.

$$k_{avg}(t) = \begin{cases} \frac{1}{T}, & \text{if } 0 \leq t \leq T \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

A moving average filter distributes the power of a sample to the neighboring samples. This results in a blurring effect and reduces high-frequency components in the signal. Figure 4.3 shows the blurring effect of a moving average filter on a simulated signal. The moving average time constant T is set to the approximate update interval of the *core* domain of 50 µs.

The original signal is plotted with the dashed line, and the filtered signal is plotted with a solid line. We can see that the filtered signal loses information on the short high-frequency peaks. The phase delay of the moving average filter was corrected in the figure.

The moving average filter provides a continuous output of filtered samples. The RAPL interface, on the other hand, provides only updates after the register values are updated,

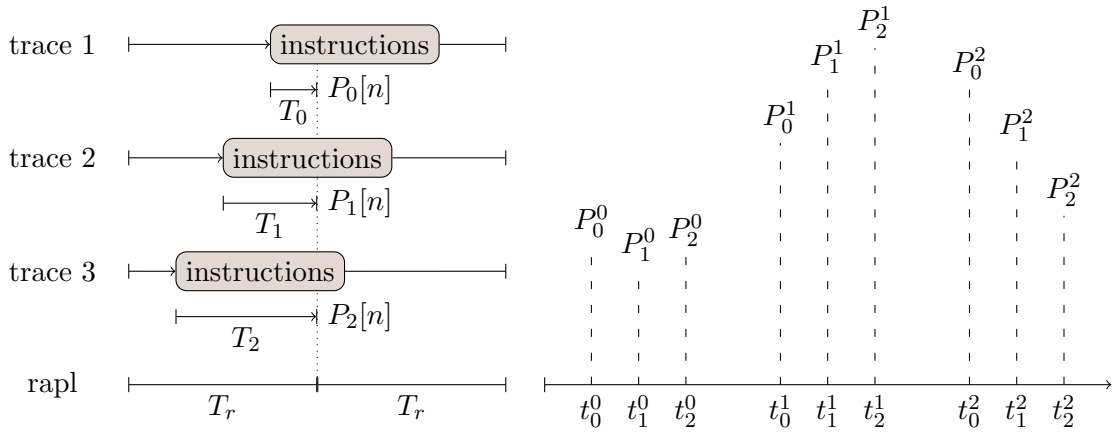

Figure 4.4: Synchronization between the traces to reconstruct a moving average filter.

i.e., the RAPL interface can be thought of as a time-discrete signal. We denote the effect of time-discrete moving average filter as a blocked average filter. The blocked average filter loses information about the data in between sample points.

Since our threat model allows us to record multiple traces (see Chapter 3), we can establish a technique to reconstruct a moving average signal from the blocked average data. Each recorded sample stores the measured TSC difference, and therefore, we can reconstruct the time points when a sample was recorded.

We discuss a synchronization technique in Section 4.2 to synchronize the start sample to a new RAPL value. We now use a similar approach, where we synchronize the start of our measurement trace to an update of the RAPL interface. However, we implement another delay after the synchronization finished, to shift the complete timeline of the RAPL update intervals in comparison with the timeline of the executed instructions.

Figure 4.4 shows this process, where we record three traces, and each of the traces gets delayed by a different time. Therefore, we shift the update interval of the RAPL interface and can reconstruct a *moving* average effect. Since the different traces are overlaid with noise, multiple traces with the same start delay can be recorded and averaged together to reduce the noise in the combined trace. To translate the shifted periods to the time of the moving average filter, the notation t_k^i is used, as shown in Equation 4.3.

$$t_k^i = T_k + i \cdot T_r \tag{4.3}$$

We need to calculate the difference energy or the average power between the RAPL interface measurements. Otherwise, the traces cannot be combined due to the different accumulating offsets. Similar to the time notation, we use the notation from Equation 4.4

to describe the RAPL readings in the figure. Where P_k describes the RAPL interface updates of the trace k.

$$P_k^i = P_k[i] \tag{4.4}$$

To calculate the time shifts for the measurement process, we upsampled the RAPL interface update interval by a factor N, i.e., we divided the update interval by an integer factor. We then can calculate the resulting shift delays with Equation 4.5.

$$T_i = \frac{T_r}{N} \cdot i \tag{4.5}$$

The shifting technique allows reconstruction to the theoretical limit of a moving average, as Figure 4.3 shows. However, we can also combine this technique with the `hlt` delay of the *measurement framework* to reduce the visible update intervals of the RAPL interface. If the `hlt` delay is used, the timestamps of the measurement samples must be adjusted accordingly. The shifting technique heavily relies on time consistency, i.e., the instructions executed inside the function must be executed at the exact same time for each trace. Since this not the case for real-world CPUs due to their complexity (see Section 2.10), we cannot reach this theoretical limit. We show a SPA attack with this technique in Section 5.3.

4.4 Extending the Measurement Resolution

In this section, we describe how to use another approach to record high-resolution side-channel information, which does not rely on time synchronization. We still use the *measurement framework* from Section 4.2. However, we extend our trigger events and built additional logic on top of it to control the resolution extension.

To improve the resolution of our side channel, we use a precise execution control method (see Section 2.11). We already use the timer interrupt as a trigger event. Thus, we adapted the *measurement framework* to support *zero*- and *single*-stepping as introduced by Van Bulck et al. [12].

Our patched SGX-SDK with our AEP callback enables us to provide logic shortly before using the `eresume` instruction to resume the enclave after a timer interrupt occurs. Figure 2.1 provides a summary of the SGX control flow transitions. To adapt *zero*- and *single*-stepping, the local APIC timer interrupt is now strictly configured in *one-shot* mode.

We developed a calibration tool to determine the *one-shot* intervals for the *zero*- and *single*-steps. This calibration tool uses an SGX enclave containing a function, with an endless loop, filled with a large sled of the four-byte long `nop` instruction. The calibration enclave is built in debug mode, enabling us to use the `edbgrd` instruction

to read the current instruction pointer of the enclave. In the first phase, we want to find the *single*-step interval. We start with a large *one-shot* interval and wait for the next timer interrupt. After the timer interrupts returned to our modified AEP callback, we compare the difference between the current enclave instruction pointer with the instruction pointer from the last AEP callback. If we observe a difference larger then four bytes, the *single*-step interval is too large, and we decrease it. On the other hand, if the difference is zero, we increase our *single*-step interval. On top of this basic logic, we build a filter. This filter prevents jumps between increments, and it adapts for outliers. If the *single*-step interval satisfies our filter and resulted in a reasonable amount of consecutive *single*-steps, we calculate the *zero*-step interval by multiplying the *single*-step interval with 0.82 This fraction was chosen empirically. To achieve a better resolution, we manually set the local APIC timer interval multiplicator to *one*. This doubles the resolution in contrast to the multiplicator of *two* used in *SgxStep*.

We also modify the interrupt handler in our kernel module. The handler is no longer a direct measurement trigger event. Instead, we introduce a flag that specifies if the interrupt should be used as a measurement trigger event or not. This flag gets reset if a measurement was triggered. This allows us to use the same timer interrupt handler as a trigger event, as a callback when the `hlt` delay expires and as a handler for the *zero*- and *single*-stepping mechanism. Only in the trigger event case, the handler needs to do actual work. In the other two, the handler simply invokes the `iret` instruction and returns to the AEP callback or the `hlt` delay invoker.

To control the *zero*- and *single*-stepping mechanism and the measurement trigger events, we implemented logic in the AEP callback. The first part of this logic is the instruction counter. The instruction counter is incremented if the accessed bit of the Page Middle Directory (PMD) of the enclave is set. This allows us to emulate information about the instruction pointer. The accessed bit of the PMD is reset before the AEP executes the `eresume` instruction. The instruction counter keeps track of the position in the enclave functions relative to the first enclave page-fault.

With the help of the instruction counter, we implement *start* triggers. *Start* triggers are stored in an array and are ordered from smallest to highest. As the name suggests, they start the logic for *zero*- and *single*-stepping mechanism at a given instruction. As soon as a *start* trigger is reached, the logic is in the running state. To count the number of AEP callbacks, we introduce the runtime counter. This counter is incremented each time the AEP callback is called if the logic is in the running state. On top of the runtime counter, we build three more triggers, which configure the behavior of the *measurement framework*.

Measurement trigger If the runtime counter is a multiple of this value, the next timer interrupt is used as a trigger event for the kernel module to record a measurement sample into the measurement buffer. The trigger event still uses the behavior of either the *symmetric* or the *asymmetric* trigger type.

```
1  void aep_callback_impl() {
2      instruction_counter += was_pmd_accessed();
3      clear_pmd_accessed();
4
5      int start       = (instruction_counter == *pstart_triggers);
6      int is_running  = start || state_running;
7
8      pstart_triggers += start;
9      runtime_counter += is_running;
10
11     int single_step = (runtime_counter % single_step_trigger) == 0;
12     int measure     = (runtime_counter % measurement_trigger) == 0;
13     int stop        = (runtime_counter % stop_trigger)        == 0;
14
15     kernel->measure = is_running && measure || start;
16     state_running   = is_running && !stop;
17
18     int interval    = step_intervals[!is_running || single_step];
19     apic_one_shot(interval);
20 }
```

Listing 4.4: The AEP callback logic to control the *zero*- and *single*-stepping mechanism.

Single step trigger If the runtime counter is a multiple of this value, the logic will perform a *single*-step to continue to the next instruction. If the logic is in the running state, the default behavior is to perform *zero*-steps to force the current instruction to be reissued.

Stop trigger If the runtime counter reaches this value, the logic leaves the running state and *single*-steps until the next *start* trigger is reached. If the last *start* trigger was stopped, the logic could stop the *zero*- and *single*-stepping mechanism since the measurement for the current trace is completed.

The AEP callback gets called after each *zero*- and *single*-step and is often executed between the *start*- and *stop*-measurement of a sample. Therefore, we reduce the measurement overhead and build the logic with as few branches and instructions as possible. Listing 4.4 shows this logic. The modulo operation to calculate the trigger flags can be replaced with an `and` operation if the three triggers values can be represented in the form of $2^N - 1$, where N is a positive integer. This reduces the energy overhead even further. This is the most general form of the AEP callback logic, and parts of it can be omitted if they are not used for a specific attack.

To reduce the noise of the overhead even further, we designed an experimental local APIC timer interrupt handler, which instead of using the Linux kernel structure directly modifies the gate descriptor inside the IDT. With this modification, we can build a low-level interrupt handler. The original interrupt entry saves all the registers and uses additional logging and functions to verify system integrity. All these instructions are

included in the RAPL energy measurements and introduce noise. We can reduce this noise by either averaging more traces together or by using the low-level handler. The handler is written in assembly, and therefore, we implement only the synchronization and the necessary measurement triggers.

The advantage of this low-level handler is that we only have a few instructions in the interrupt path as Listing 4.5 shows.

```
asm_timer_interrupt_handler:
    push %rax
    ; ack apic eoi
    mov apic_base_ptr(%rip), %rax
    movl $0, 0xb0(%rax)
    ; check trigger flag
    mov trigger_flag_ptr(%rip), %rax
    cmpb $0, (%rax)
    jne asm_measurement_triggered
    ; leave and interrupt return
    pop %rax
    iretq

asm_measurement_triggered:
    ; record measurement sample here
    pop %rax
    iretq

asm_aep_callback:
    mov $3, %rax
    ; call C aep callback here if necessary
    enclu
```

Listing 4.5: Low-level assembly interrupt handler.

In the listing, only the relevant instructions are in the default path where we do not trigger a measurement. The `apic_base_ptr` is a pointer to the memory-mapped local APIC. We acknowledge the interrupt by writing to a specific memory address in the mapped area (see Section 2.9). The `trigger_flag_ptr` points to a variable inside the `mmaped` *measurement framework* control structure and signals if a measurement sample should be recorded. The assembly AEP callback then simply invokes the `eresume` instruction and returns the control flow to the enclave. One caveat here is that all the watchdogs of the operating system must be disabled, otherwise, the operating system will detect a lockup on the victim core since we no longer update the watchdog timer in the timer interrupt handler.

4.5 Precise Instruction Finding

In our attack, we want to precisely single step to a given instruction and then use our *measurement framework* to record side-channel information. We denote this process as targeted *zero*- and *single*-stepping. In contrast to the blind *zero*- and *single*-stepping,

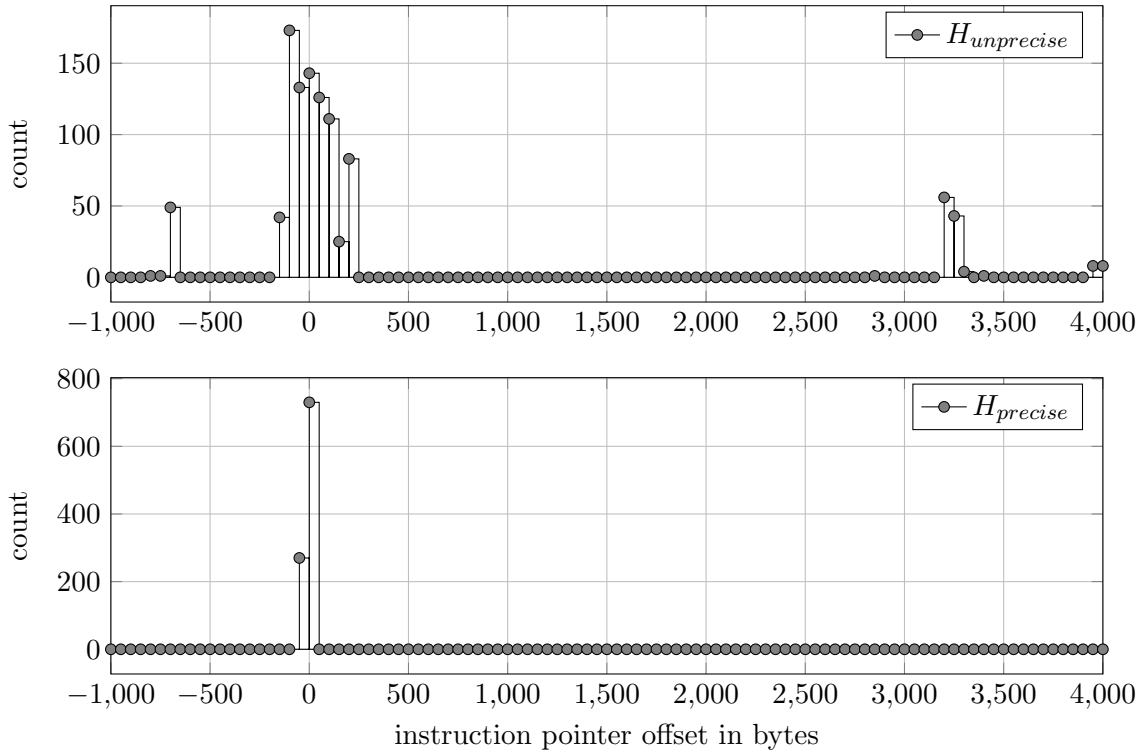

Figure 4.5: Comparison of the instruction jitter between precise and unprecise mode.

we know the instruction which we want to record and can, therefore, skip the instruction which we are not interested in. Targeted *zero*- and *single*-stepping has a considerable performance benefit if the code has thousands of instructions, which would increase the time to record a measurement trace dramatically when using the blind method.

We described in Section 4.4 how we use the *start* triggers to start the *zero*- and *single*-stepping logic. We discuss in this section how we can enforce that the instruction counter of the AEP callback is precisely counting the instructions with low jitter, i.e., how we can optimize our *single*-steps to be accurate.

Since the execution time of an instruction depends on the instruction itself, each instruction would need its local APIC timer *one-shot* interval, and we would need to know the instruction currently being executed in advance. Since this is not feasible for the attacker, we developed a precise *single*-stepping mechanism. If we want to execute a *single*-step, but the *one-shot* interval is either to larger or too small for the current instruction, a *zero*-step or a *multi*-step can occur. We denote these *single*-stepping errors as *false single*-step. If we did not use the precise *single*-stepping mechanism, the *false single*-steps would add up, and we would only measure in the surrounding of our target instruction.

```
1  void aep_callback_impl() {
2      if (single_step_flag && !was_pmd_accessed()) {
3          increment++;
4          apic_one_shot(minimal_interval + increment);
5          return;
6      }
7      increment = 0;
8      /* ... */
9      single_step_flag = !is_running || single_step;
10     int interval     = step_intervals[single_step_flag];
11     apic_one_shot(interval);
12 }
```

Listing 4.6: The modified AEP callback logic to implement precise *single*-stepping.

In order to establish this precise *single*-stepping technique, we first define the minimal *single*-step *one-shot* interval for all instructions. This is the interval of the one-byte long `nop` instruction because it is among the smallest and fastest instructions in the Instruction Set Architecture (ISA). We still use the page table flags to distinguish the *zero*-steps from the *single*-steps. If we now trigger a *single*-step, we set a flag to signal that the next instruction should be single stepped, and then set the next *one-shot* interval to the minimal *single*-step interval. Upon the next AEP callback entry, we check if this flag is set. If the flag is set and the accessed bit of the PMD is not set, then we add an increment to the minimal *single*-step interval and trigger the *one-shot* mode again and execute an early return from the AEP callback.

With this technique, we can minimize the *false single*-steps since we slowly detect the real interval to *single*-step the instruction. If the precise *single*-stepping mode is used, the measurement time might be extended, due to the retries. We can either use the TSC difference to normalize this effect or disable precise *single*-stepping if the logic is in running mode.

Figure 4.5 compares the precise *single*-stepping mechanism against the simple *single*-stepping where no access flags are checked. In the figure, the 100000^{th} instruction is targeted, and the x-axis shows the jitter around the target instruction address in bytes. Due to the retry mechanism, the recording of a measurement trace took 6 ms longer. However, we see a significantly reduced jitter when using precise *single*-stepping.

4.6 Classes of Victim Algorithms

In this section, we classify different types of algorithms a victim could use inside an enclave and how we need to measure to extract the secret information from the enclave. We differentiate between the following victim algorithm types.

***Asymmetric* Algorithm** This algorithm type uses the secret information inside a branch condition, and the resulting branches have different instruction counts.

***Symmetric* Algorithm** This algorithm type uses the secret information inside a branch condition similar to the asymmetric algorithm. However, the branches of the condition have the same instruction count. The instructions themself can be different. Only the number of instructions is essential.

***Constant* Algorithm** The secret information is not used inside a branch. For each of the two cases, the same instructions are executed, and the secret information is encoded in the data of the instructions.

We assume that each of these algorithms is embedded in large binaries with thousands of instructions, and we, therefore, use targeted *zero*- and *single*-stepping, as described in Section 4.5 to attack the algorithm. If the algorithm uses only a few instructions, we can use blind *zero*- and *single*-stepping, as we show in Section 5.4 to extract the secret information processed in the algorithm. Each of these algorithm types must be handled differently by the *measurement framework*.

The *asymmetric* algorithm is the hardest to mount targeted *zero*- and *single*-stepping on because we must classify the secret information during the measurement phase. Otherwise, we do not know which instruction to measure next. We also cannot postpone the classification after the measurement phase, since the samples to measure the instruction containing the secret information, would grow exponentially with the length of the secret information. If the instruction counts of the two branches have a common divisor, this effect is reduced since the measurement points can overlap.

The *symmetric* algorithm can be targeted without the classification in the measurement phase. Both of the branches have the same number of instructions, and therefore we can simply measure each N^{th} instruction, where N corresponds to the instruction count of the cases. Certainly, we must target an offset at which both branches have either a different instruction or where the instruction encodes the secret information.

To target the instructions inside a *constant* algorithm, the same approach as by the *symmetric* algorithm is used. The branches for the two cases have the same length and the same instructions. The secret information is encoded in the operands of the instructions, similar to the constant time cryptographic algorithms from Section 2.6. The energy signatures of different instructions are more distinct than the energy signature of different operands. Therefore, the number of traces needed to reconstruct the secret information is higher in comparison to the *symmetric* algorithm.

For the *symmetric* and *constant* algorithm, we can record all of the occurrences of the secret information in one trace, and therefore use fewer traces to reconstruct the secret information. In an attack on an *asymmetric* algorithm, we need to record more traces to classify a single part of the secret information and therefore need more overall traces.

4.7 Debug Profiling

In this section, we detail how to generate *start* triggers. For the targeted *zero*- and *single*-stepping approach, we assume that the source code of the enclave is available. This implies that the enclave does not receive encrypted source code over a trusted server by generating a public and private key pair an sending the public key to the server with some authentication information as described by Aumasson et al. [17].

The first possibility to generate a start trigger is to count instructions from a given page-fault manually. This approach works for small examples, as we show in Section 5.4, where we first blindly *zero*- and *single*-step and then use each sixth instruction to classify the key. Blindly *single*-stepping to determine patterns and then start targeted *single*-stepping can increase the performance dramatically as we show in Section 5.4, where we observe an overhead of approximately six times in the toy example.

The overhead can be approximated by the number of instructions inside the function N and the number of instructions targeted with targeted *zero*- and *single*-stepping N_t. Equation 4.6 shows the relation between the execution time of blind *zero*- and *single*-stepping in comparison with the target approach.

$$\frac{t_{blind}}{t_{targeted}} \approx \frac{N}{N_t + (N - N_t) \cdot \alpha} \tag{4.6}$$

Equation 4.7 shows how to calculate the scaling factor α. Where t_{ss} is the time needed for a *single*-step, t_{zs} is the time needed for a *zero*-step, and N_{zs} is the number of *zero*-steps per instructions.

$$\alpha = \left(\frac{N_{zs} \cdot t_{zs}}{t_{ss}} + 1 \right)^{-1} \tag{4.7}$$

If the calibration tool from Section 4.4 is used, the time for a *zero*-step is calculated by multiplying the time for a *single*-step by 0.82 and therefore α can be simplified to Equation 4.8.

$$\alpha = (N_{zs} \cdot 0.82 + 1)^{-1} \tag{4.8}$$

The manually counting approach is not feasible for large binaries, or if the enclave function uses loops. Therefore, we develop a profiling mechanism. We can read a specific value from the enclaves SSA into the *debug* field of the measurement sample, by passing an address to our measurement kernel module and using the `access_process_vm` Linux kernel function to read the enclave's memory, as used by *SgxStep* [12]. Since in the SSA are all registers stored during a context switch, we have access to the current instruction pointer. If a debug copy of the enclave is available, or the source code of the enclave can be used to build a debug enclave, we can use this debug enclave to record the *start*

triggers for the *measurement framework*. We denote such a debug enclave, where we can execute the same code but without the secret information as a trigger oracle.

To generate *start* triggers from the oracle, we specify the address of the target instruction. We then precisely *single*-step each instruction. Each time the kernel module timer interrupt handler is invoked, we read the enclaves instruction pointer and check if we are currently at the desired instruction. If we hit the instruction, we store the instruction counter as a new *start* trigger and continue the search. We store the recorded *start* triggers in a file, which can later be loaded by the *measurement framework*.

Since the configuration of the *zero*- and *single*-step intervals is complex, we also build the reverse mechanism, where we record the instruction pointer into each measurement sample. Thus, we can verify that the measurements are taken from the desired instruction.

4.8 Power and P-State Monitoring

In the measurements of our side channel, we exploit short term changes in the power consumption or the core voltage to extract secret information. Nevertheless, we want to establish the same measurement conditions for each of the samples and traces recorded, so we can post-process them to recover the secret information.

In order to establish equal conditions for each of the samples in a trace, we fix the P-state of the cores and disable the Intel *Turbo Boost Technology*. This mitigates P-state transitions and jumps in energy consumption and the core-voltage operation point during the different phases of an attack. To enable software P-state requests, Hardware Controlled Perfromance States (HWP) must be disabled, and the operating system must use a power governor accepting user-requested P-states.

We cannot control external factors, like thermal heating, so we implement additional monitoring for distorted traces. First, we monitor the current P-state for each sample we record with the P-state field. As we mentioned in Section 4.2, reading the current P-state introduces no overhead, since we already read the core voltage from the same MSR. In practice, we did not observe P-state changes after configuring the hardware to use the requested P-states, but we may observe a drop if thermal throttling starts or the TDP is reached.

In addition to the monitoring of the samples, we also look at the mean trend of traces during the *measurement process*. We calculate the mean over the energy and core-voltage fields of the samples inside a trace. We then repeat this process for each trace and plot the results over the trace number. If all the traces are recorded under the same conditions, the means should be the same for each trace, and the plot should show a straight line. In practice, we observe different behavior of the first few traces when starting the *measurement process*. We remove these traces and only use the traces where the means lie within a margin of each other.

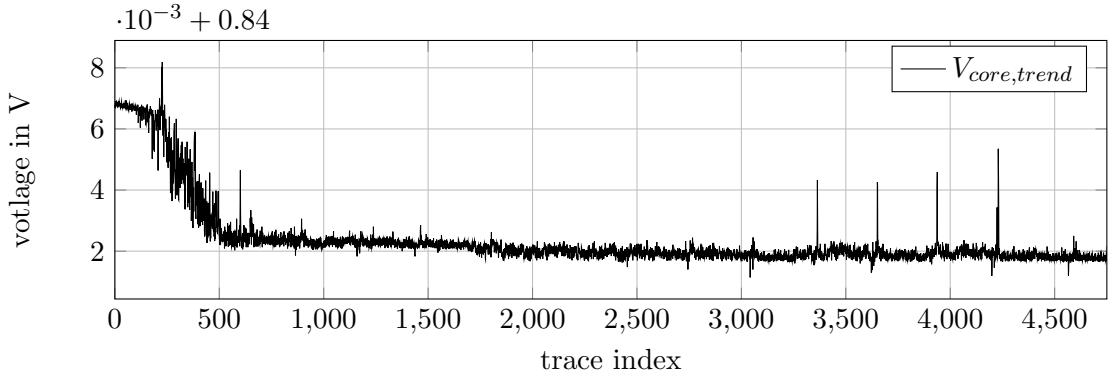

Figure 4.6: The mean trend of the core-voltage domain over the measurement traces.

Figure 4.6 shows an example of the mean trend of the core-voltage field over the measurement from Section 5.4. We can see that the first 500 traces are not recorded under the same conditions and therefore remove them from the post-processing step. In the figure, we can also observe traces with spikes around trace index 3700. This technique is also useful if a background process starts on the general-purpose core and distorts the measurements.

Chapter 5

Attack Evaluation

In this section, we use the measurement framework and the *zero-* and *single*-stepping extension (see Sections 4.2 and 4.4) to exploit power side-channel leakage on cryptographic algorithms.

We describe the configuration of the mbedTLS library for our attacks in Section 5.1. Section 5.2 describes the system we use in the evaluation of our attacks. We then use the mbedTLS library in Section 5.3 to use the time-based reconstruction technique to exploit SPA leakage of the *square-and-multiply* algorithm. In Section 5.4, we attack a toy example to show that we can distinguish the energy signature of different branches inside the same cache line. In addition, the branches use the same instruction except that one of the operands is different. We then advance the attack on the *square-and-multiply* algorithm from the mbedTLS SPA example to an instruction-level resolution and target an AVX instruction inside one of the key bit paths in Section 5.5.

We show how the configuration options of the measurement framework can be used to target each of the above algorithms without the need to change the framework. Also, we show the influence of the different parameters for the same attack target.

5.1 Victim Setup and MbedTLS Configuration

In this section, we discuss the setup for the victim algorithms we attack with our *measurement framework*. All of the algorithms used in Sections 5.3 to 5.5 are placed inside an SGX enclave. Figure 5.1 shows the general setup of our attacks. The secret information is always stored in the secure enclave memory. The victim exposes a defined API to the victim, which can be queried arbitrarily (see Chapter 3).

In Sections 5.3 and 5.5, we use the mbedTLS library as a victim. We adopted the mbedTLS library configuration to satisfy the restriction of an SGX enclave (see Section 2.8). We use the mbedTLS library version *2.13.0* for our attacks.

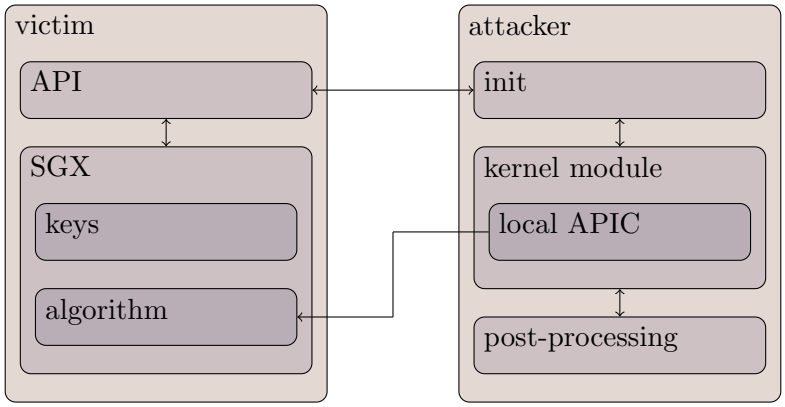

Figure 5.1: Block diagram of the attacker and victim interaction.

We disabled all the operating system features like file input and output operations. The only remaining output functionality was the `printf` function, which we routed to an *ocall*.

The RSA implementation uses an optimized version of the binary exponentiation algorithm (see Section 2.6). To achieve direct leakage of the key over a conditional branch, we set the moving exponentiation window length to one. Liu et al. [45] showed that if an attack can be mounted on windows with a length of one, the attack can be extended to arbitrary windows sizes.

To enhance the security of the algorithm, the mbedTLS library uses exponent blinding and fault/glitch checking. This checking is achieved by encrypting the message again after decrypting it and test if the result matches the input message. Since these counter mechanisms increase either the number of traces needed or the execution time of the algorithm itself, we did not include these mechanisms in our attacks. We used the modulo exponentiation function of mbedTLS directly. The result is a *square-and-multiply* algorithm that is capable of handling big integers and uses a branch condition with the secret information inside the branch.

5.2 Test System

The following attacks were executed on an *Intel(R) Xeon(R) CPU E3-1275 v5 @ 3.60GHz* CPU. The system is running *Ubuntu 18.04.3 LTS* with kernel version *5.0.0-15-generic*. We set the following kernel boot parameters:

`nox2apic` Allows us to memory map the APIC.

`apic=debug` Prints advanced information if the IDT is corrupted.

`nmi_watchdog=0` Disables the `nmi_watchdog`, otherwise, it would detect that the victim core is not handling timer interrupts, and try to recover by switching to a different

timer interrupt source. Since we hijacked the interrupt handler, we do not want this behavior.

`maxcpus=2` Disables all the cores except for two. So we have one core as general-purpose core and one of the cores as victim core.

`isolcpus=1` Removes the core number one from the scheduling list and allows us to use it as victim core.

`intel_pstate=no_hwp` disables Hardware Controlled Perfromance States (HWP) and enables us to set the P-state directly and without dynamic P-state transitions.

The remaining parameters are reused from the *SgxStep* framework [12].

5.3 SPA Attack

In this section, we use SPA with our *measurement framework* to reconstruct parts of a mbedTLS RSA key.

Implementation The target for this attack uses the same mbedTLS configuration, as described in Section 5.1. We use a 2048-bit RSA modulus with a 512-bit key. The larger the modulus size, the longer the mbedTLS algorithm takes to calculate the multiplications of the intermediates. Therefore, the time between key bits is increased. The key consists of different blocks of zeros and ones, as shown in Equation 5.1. The *measurement framework* is configured to trigger the victim API and record samples over the invocation.

$$D = \texttt{0xffffffff00000000ffffffff00000000ffff0000ffff0000ff00ff00ff00ff00}\ldots \\ \texttt{00000000000000000000000000000000ffffffffffffffff0000000000000001} \tag{5.1}$$

Configuration We use the technique from Sections 4.2 and 4.3 to increase the resolution of the RAPL interface. We set the `hlt` delay to 35 µs, so we reduce the seen update interval of the *core* RAPL domain to 15 µs. To reconstruct the moving average effect from Section 4.3, we delay the invocation of the victim API with a uniformly distributed delay between 0 µs and 15 µs.

The P-state was reduced to 20 % of the maximum P-state to increase the duration of the victim function. We observe a trade-off between the number of samples per trace, i.e., the duration of the invocation and the number of traces needed to reconstruct the key parts when reducing the P-state, since the energy consumption also drops and, therefore, the leakage effect is also decreased.

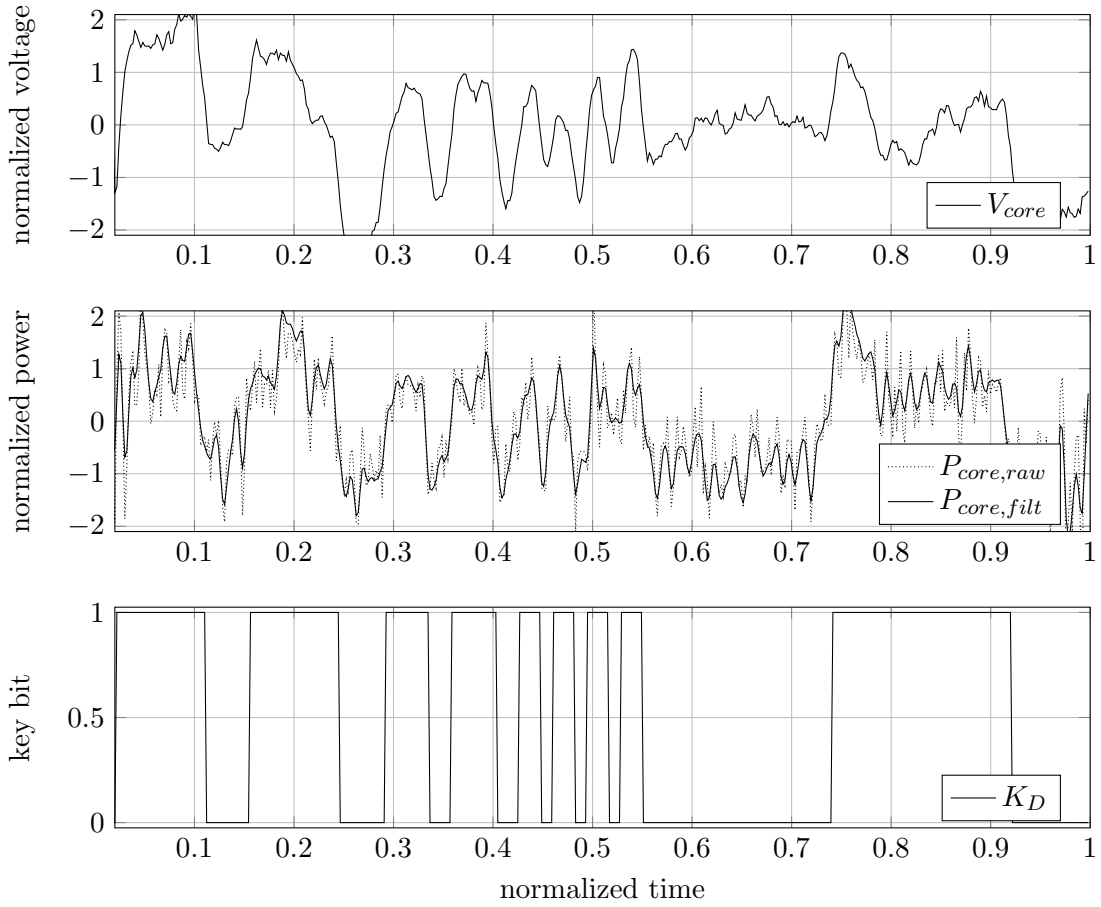

Figure 5.2: RSA key leakage with a classical SPA attack.

Results We record 500,000 traces over 12 hours and apply the following post-processing technique. Due to fluctuations in the execution speeds of the CPU, some of the traces take less time than others. To correct this effect, the time-stamps are normalized to a range between zero and one, where one represents the end of the algorithm shifted by the start delay, and zero represents the start of the recording, including the shift delay to reconstruct the moving average effect. We then apply a linear interpolation to increase the number of samples of the traces to 2000. We need to apply this step since the recorded traces contain a different number of samples per trace. The resulting traces have all the same length and are interpolated to equidistant time stamps between zero and one.

We then use a moving average filter with a length of two on each of the traces to distribute the sample energy between the neighboring samples and then combine the traces. As a combination method, we first apply a per trace normalization and then use the median function to combine the traces. The resulting trace represents the normalized power

consumption or the normalized core voltage over the normalized time.

Figure 5.2 shows the resulting traces for the *core* RAPL domain and the core voltage. We apply a moving average filter of length three to the result of the *core* RAPL domain to reduce the noise from the trace.

We can see that the core voltage and the *core* RAPL domain leak the RSA key blocks of the private key (see Equation 5.1). We also see the effect that the algorithm approximately takes twice the time to calculate a one bit in comparison to a zero bit. Therefore, we adjust the timing of the key plot from Figure 5.2 to match the timing of the currently processed key bit. We can distinguish 8-bit blocks in the core-voltage trace, but the core voltage is slowly fluctuating, and we cannot distinguish the last 64-bit block of ones. In the *core* RAPL trace, we see no fluctuation but cannot distinguish the last pair of alternating 8-bit blocks. However, we can distinguish the remaining key blocks.

We conclude that we can mount classical SPA attacks from software on RSA algorithms with some remaining limitations. The resolution of the bit blocks could be extended by increasing the `hlt` delay or by using uncacheable memory, as we discuss in Section 7.3. If the victim uses a slower implementation or an implementation with a more distinct key leakage, SPA attacks become more realistic.

5.4 Attack on a Symmetric Algorithm

In this section, we attack a *symmetric* algorithm (see Section 4.6). A *symmetric* algorithm uses the secret information inside a branch condition, but the two resulting branches have the same number of instructions. In addition to the same instruction count, we decided to enhance the attack by using the same instructions in both of the branches, only differing in one register operand.

Implementation The toy example implements a *square-and-multiply* like algorithm (see Section 2.6). Usually, this algorithm needs a *big* integer implementation to handle arbitrary-length integers. We keep the example small and use 256-bit AVX registers. A caveat of the toy example is that it is not reversible, i.e., we cannot apply the *encrypt* and then *decrypt* function to receive the original message.

We protect the algorithm by placing it inside an SGX enclave and only provide a defined API to encrypt a message. We place the key inside the static data segment of the enclave. In a real algorithm, the private key is managed differently and not placed inside the binary as constant. In our example, only the fact that the key is stored in the enclave memory is important. Listing 5.1 shows the *C* code of the toy example.

The toy example uses the *Labels as Values* extension of the GNU Compiler Collection (GCC), to generate a jump table. We compile the code with the maximum optimization level, so the jump table is only computed once. With the jump table, it is possible to generate two branches with the same instructions count directly from *C*.

```
uint32_t key[]   = {1,0,1,1,0,0,1,0,1,0,2};
uint32_t init[8] = {1,1,1,1,1,1,1,1};

void toy_example(uint32_t in_out[8]) {
    const void *jmp_table[] = {
        &&case_zero,
        &&case_one,
        &&case_end
    };
    uint32_t *pkey = key;
    asm volatile("vlddqu (%0), %%ymm0"::"r"(init));     // y = init
    asm volatile("vlddqu (%0), %%ymm1"::"r"(in_out));   // x = in_out
    goto* jmp_table[*pkey++];

case_zero:
    asm volatile("vpmuludq %ymm0, %ymm0, %ymm0");       // y = y^2
    asm volatile("vpmuludq %ymm2, %ymm1, %ymm0");       // t = x*y
    goto* jmp_table[*pkey++];

case_one:
    asm volatile("vpmuludq %ymm0, %ymm0, %ymm0");       // y = y^2
    asm volatile("vpmuludq %ymm0, %ymm1, %ymm0");       // y = x*y
    goto* jmp_table[*pkey++];

case_end:
    asm volatile("vmovdqa %%ymm0, (%0)"::"r"(in_out)); // in_out = y
}
```

Listing 5.1: *C* code of the toy example.

We terminate the key with a sentinel, similar to *C* strings, so we know when to stop the toy algorithm. To handle the sentinel, i.e., the end of the algorithm, we simply implement and additional case in the jump table. To prevent the compiler from inlining the key, the variable is not declared static. We define three labels for the *zero*, *one*, and *end* cases. The next case is called by indexing the jump table with the current key bit and perform an indirect jump. We use the AVX `ymm0` to `ymm2` registers for the data and the calculation intermediates. We use the `ymm0` as our current sate or intermediate of the algorithm. In the `ymm1` register, we store the passed input argument of the function, and finally, the `ymm2` register is used as a dummy register to store the result in the *zero* case.

The algorithm then loops through the bits of the *key*, which we represent as unsigned 32-bit integers, so we do not need to shift out the current bit and can instead use a distinct address to access the next bit. If the algorithm jumps to the *zero* case, the state register `ymm0` is squared. We use the `vpmuludq` instruction for squaring and multiplying. This instruction interprets the AVX register as packed unsigned 32-bit integers and preforms pice-wise multiplications. After the `ymm0` register is squared, multiplication with the input register `ymm1` is performed, but the algorithm discards the result of the

multiplication by storing it in the dummy register `ymm2`. The difference to the *one* case is that there the results of the multiplication gets stored in the state register `ymm0`. When the algorithm reaches the sentinel, it stores the state register into the input pointer and returns from the function. The complete assembly code of the two branches for the *one* and *zero* case is 46 bytes large and, therefore, it fits inside a single cache line, which is typically 64 bytes large.

```
1 case_one:
2 vpmuludq %ymm0,%ymm0,%ymm0
3 vpmuludq %ymm0,%ymm1,%ymm0
4 movslq   -0x4(%rax),%rdx
5 add      $0x4,%rax
6 mov      -0x28(%rsp,%rdx,8),%rdx
7 jmpq     *%rdx
```

Listing 5.2: Assembly of the *one* case.

```
1 case_zero:
2 vpmuludq %ymm0,%ymm0,%ymm0
3 vpmuludq %ymm2,%ymm1,%ymm0
4 movslq   -0x4(%rax),%rdx
5 add      $0x4,%rax
6 mov      -0x28(%rsp,%rdx,8),%rdx
7 jmpq     *%rdx
```

Listing 5.3: Assembly of the *zero* case.

Configuration In this paragraph, we describe the configuration of the *measurement framework* from Sections 4.2 and 4.4. We assume that we have no access to the source code of the victim and perform blind *zero*- and *single*-stepping to extract the secret information. Section 2.11 describes the technique used to start the attack by finding the first code page after the victim's API is called. We use the page accessed flags to determine which page is accessed after the call (see Section 2.11).

We do not know when the processing of the key starts, so we *zero*-and *single*-step each instruction in the function until we return to our caller. To measure continuously, we use the in Section 4.2 described *symmetric* trigger, which stops the previous measurement and starts a new one, after each trigger event. As a trigger event, we use the in Section 4.4 described *zero*- and *single*-stepping mechanism. To determine the intervals of the local APIC timer interrupt for the *zero*- and *single*-steps, we use the calibration tool on our test system described in Section 5.2. We reduce the CPU frequency to 75%.

The AEP callback logic was configured to trigger a *single*-step and a measurement after N *zero*-steps. We did not use the `hlt` delay nor the stop trigger and set the start trigger to the first instruction, so we measure each instruction from the first page fault. The configured attack features now only one real parameter. The others are provided by the calibration tool for the CPU running the attack. We can influence the measurement duration per instruction with the parameter N. The RAPL register has a defined update rate as described in Section 4.1, so we choose N larger than the minimum value of 186 to receive updates in all the RAPL domains. If the value is below this margin, we do not receive updates in the *package* and *dram* domain on the test system.

Results In the first run, we set N to 188 and measured 96,000 traces. This N is near the minimum value such that the *package* domain receives one update on the test system. The measurement on the test system took 8.11 hours.

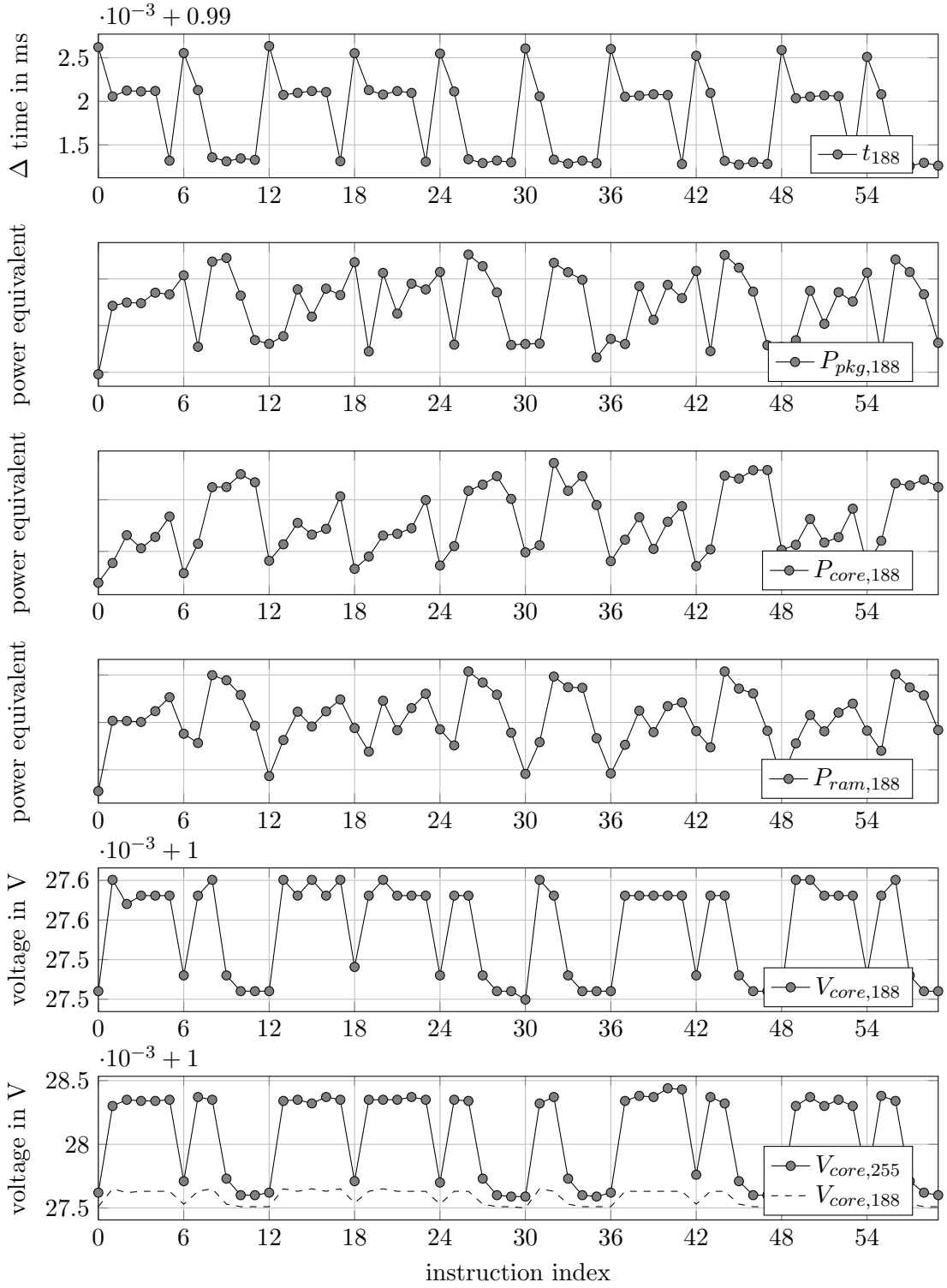

Figure 5.3: Results of the RAPL domains and the core voltage.

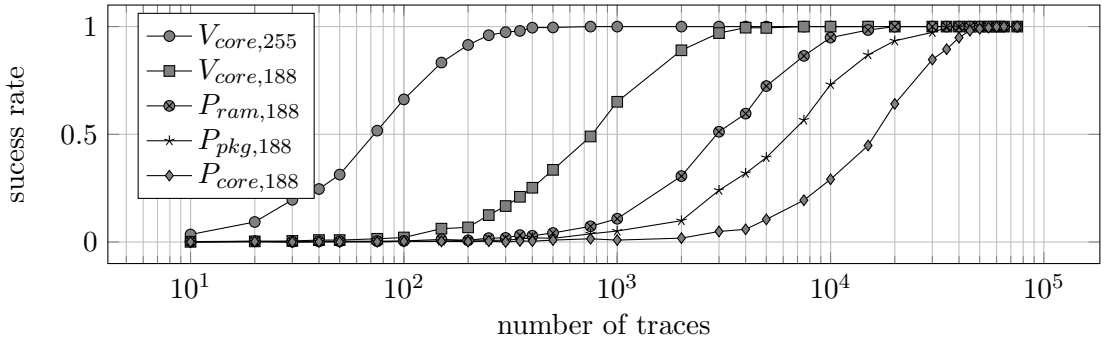

Figure 5.4: The classification trend over the number of traces recorded.

Due to the blind *zero*- and *single*-stepping technique, each of the instructions inside the API function represent a sample inside the measurement trace. We describe in Section 3.2, that multiple invocations of the same victim function result in the same executed instructions. Therefore, we can remove outlier traces, which do not contain the same number of samples as the median of all the traces.

From the 90,000 traces of the first measurement, around 2200 traces were removed. We observe outliers with only one or two samples difference. We post-process the remaining traces by calculating the mean for each field of a sample over all the traces, resulting in a single trace where each sample represents one instruction inside the victim function.

Listings 5.2 and 5.3 shows that the branches consist of six instruction each, the same period is observable in the measurement results from Figure 5.3. In the figure, we zoomed into the interesting part and cut away the entry and exit of the function.

We see a repeating pattern in all the energy domains and the core voltage. If we compare the pattern to the key from Listing 5.1 (1,0,1,1,0,0,1,0,1,0), we see two different patterns with the length of six samples, and the pattern matches the key.

We also looked at the effect of setting the parameter N to 255 and reran the experiment with only 20,000 traces. As visible from Figure 5.3, the voltage leakage increased in comparison with the core-voltage trace of the first experiment.

In the first attack, we use blind *zero*- and *single*-stepping to record each instruction inside the enclave function. We observe that the pattern for *one* key bit and the pattern for a *zero* key bit can be distinguished by only looking at one specific instruction. This enables us to use targeted *zero*- and *single*-stepping. To determine the effectiveness of our attack, we randomly selected a subset of the recorded traces and only use this subset in the post-processing algorithm.

$$k_i = \mathcal{C}(x, i) = \left(x_i > \frac{min(x) + max(x)}{2} \right) \tag{5.2}$$

Domain	Instructions	N	Traces	Success Rate	Duration
package	6 (blind)	188	60000	99.7%	203.96 h
core voltage	6 (blind)	255	150	99.4%	41.65 min
package	1 (targeted)	188	40000	99.5%	29.16 h
dram	1 (targeted)	188	20000	99.7%	11.67 h
core	1 (targeted)	188	55000	99.6%	32.09 h
core voltage	1 (targeted)	255	350	99.0%	16.55 min

Table 5.1: Attack times for a 2048-bit key within the toy example.

We then post-process this subset and apply the classification algorithm from Equation 5.2 on the resulting post-processed trace x. The post-processed trace only contains the targeted instructions. We use every fourth instruction for the RAPL domains and every sixth instruction for the core voltage, so we have to decide which side channel we use in advance.

We use a simple threshold comparison to determine the key bit. The threshold is the midpoint between the maximum and minimum of the used samples. We repeat the random selection process and the classification algorithm a thousand times and calculate the success rate at which we correctly classify the complete key. Figure 5.4 shows how the success rate changes over the number of traces used.

We observe that the core-voltage side channel for N set to 188 needs one order of magnitude fewer traces than the RAPL domains recorded with the same parameter N. If we increase the parameter N, we amplify the leakage in the core-voltage side channel, so we see a total reduction of two orders of magnitude in comparison with the RAPL domains.

With the given success rates and the bit timings, we calculate the time needed to reconstruct a complete 2048-bit long key within the toy example. Table 5.1 shows the resulting attack times. Hence, we can reconstruct a complete key with a success rate of 99 percent in under 17 minutes when using the core-voltage side channel and an increased parameter N of 255.

From the attack duration, we also see the benefit of targeted *zero*- and *single*-stepping over the blind approach. If we target all instructions in these cases, we observe an overhead of factor 5.87 for the core-voltage domain, if we account for the different trace counts in Table 5.1. If we use Equation 4.6, we obtain an approximated overhead of factor 5.81

Van Bulck et al. used *SgxStep* [13] to classify instruction based on their interrupt latency. The latency between the interrupt arrival and the interrupt handling depends on the instruction executed. We also see this effect in Figure 5.3 in the Δ time trace. To not include this leakage into the power calculations from Equation 2.7, we did not calculate the power in the figure, instead, we used only the RAPL differences and denoted them

as power equivalents.

5.5 Attack on an Asymmetric Algorithm

In this section, we configure the *measurement framework* to attack an *asymmetric* algorithm. We use the mbedTLS library as a target and reuse the configuration from Section 5.1. The victim is again located inside an SGX enclave with the keys lying inside the enclave's memory. Since the mbedTLS library is not constant time (see Section 2.6), conventional timing attacks can be mounted on the implementation. We focus on extracting the secret information with our *zero*- and *single*-stepping extension and the RAPL or core-voltage side channel instead of timing attacks.

Implementation We mount targeted *zero*- and *single*-stepping on the algorithm since the library supports arbitrary length integers and is a full-grown cryptographic algorithm in comparison to the toy example from Section 5.4, therefore blind *zero*- and *single*- stepping is not feasible. We show the performance overhead of blind *zero*- and *single*- stepping in Equation 4.6. The two branches of the *asymmetric* algorithm contain different instructions and different instruction counts.

```
1  void mbedtls_mpi_exp_mod(/*...*/) {
2      // ...
3      mbedtls_mpi *p;
4      while( 1 ) {
5          if( bufsize == 0 ) {
6              if( nblimbs == 0 )
7                  break;
8              nblimbs--;
9              bufsize = 64;
10         }
11         bufsize--;
12         ei = (E->p[nblimbs] >> bufsize) & 1;
13         if( ei == 0 && state == 0 )
14             continue;
15         if( ei == 0 && state == 1 ) {
16             p = X;
17             goto mul_or_square
18         }
19         state = 1;
20         p = Y;
21         MBEDTLS_MPI_CHK( mpi_montmul( X, X, N, mm, &T ) );
22     mul_or_square:
23         MBEDTLS_MPI_CHK( mpi_montmul( X, p, N, mm, &T ) );
24     }
25     //...
26 }
```

Listing 5.4: *C* code of the mbedTLS library (rearanged to fit assembly output).

To distinguish the key bit inside the branch condition, we do not attack the branch condition directly. Instead, we attack a target instruction with a more distinct energy signature than the branch itself. We target an AVX instruction inside the Intel `fast_memset` implementation. Which replaces the standard *libc* `memset` implementation inside SGX. The AVX instruction is a nearly fixed offset away from the branch condition in the *one* case. Listing 5.4 shows the binary exponentiation algorithm, where the `memset` function is called inside the `mpi_montmul` function. We use the same offset for each key bit and, therefore, will target the AVX instruction in the *one* case, but in the *zero* case, we do not see the same energy signature since another instruction is at this offset. Listings 5.5 and 5.6 shows the disassembly of the two targets. The arrows mark the target instructions.

```
1    leaq          0x0001bc50,%rsi
2  ->vmovq         %r9,%xmm0
3  ->vpbroadcastd  %xmm0,%ymm0
4  ->cmpq          $0x80,%r11
5    ja            0x1b080
```

Listing 5.5: Target instructions of the *one* case.

```
1    addq     $8,%rsp
2  ->jmp      _intel_fast_memset.V
3  ->endbr64
4  ->jmp      __intel_avx_rep_memset
5    endbr64
```

Listing 5.6: Target instructions of the *zero* case.

The *square-and-multiply* algorithm implementation of mbedTLS skips leading zeroes of the exponent and has, therefore, a key dependent setup phase. The big-integer multiplication implementation also takes a different number of instructions depending on the data of the multiplication. Furthermore, the selection of the exponent key bit also uses different instructions. All these factors add up and lead to a data-dependent offset to the target instructions.

Configuration Since we assume a known assembly attack, we could reconstruct the exact offset, if we know the plain text and mount the same attack as we describe here on the condition to skip leading zeros. With the number of leading zeros and the plain text, we can iteratively reconstruct the intermediates and circumvent the non-fixed multiplication and bit selection process.

It is possible to calculate the offsets from hand, but since we have the enclaves assembly, we instead build a trigger oracle to precisely determine the next *zero*- and *single*-stepping trigger. Equation 5.3 shows the oracle we need for the mbedTLS library implementation.

$$t_i = \mathcal{O}(i, \mathcal{K}_{i-1}, \mathcal{M}, \mathcal{N}, \mathcal{L}) \tag{5.3}$$

The oracle depends on the previously reconstructed key bits $\mathcal{K}_{i-1} = (k_0, \ldots, k_{i-1})$, the known plaintext $\mathcal{M}$, and the public modulus $\mathcal{N}$, the current index i and due to the setup phase the number of leading zeros $\mathcal{L}$. The oracle gives us the next target offset t_i where we need to *zero*- and *single*-step to record the samples $\mathcal{S}_{t_i}$. Equation 5.4 then

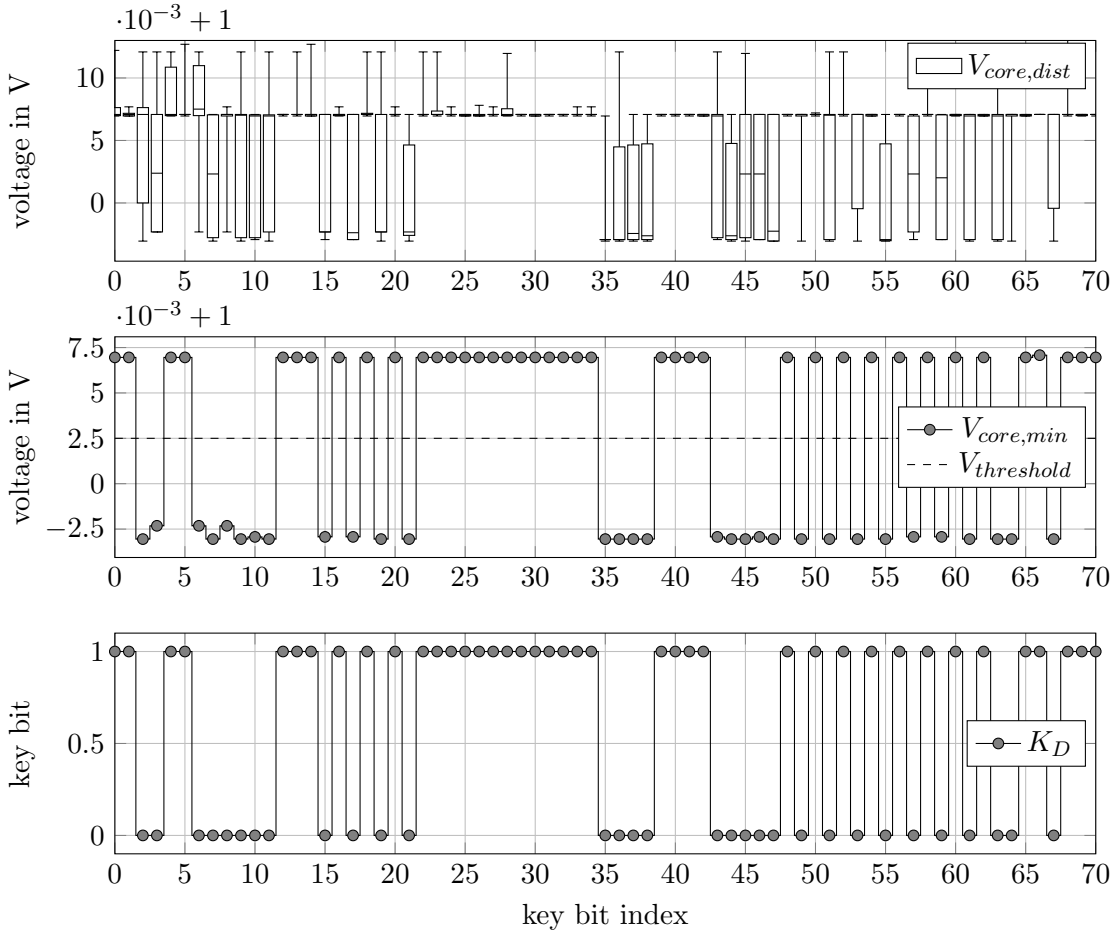

Figure 5.5: Reconstruction of the RSA key from the voltage distribution.

uses these samples to classify the next key bit k_i with a given classification algorithm $\mathcal{C}$. Listing 5.7 shows the complete algorithm to reconstruct the victim's key iteratively where $\mathcal{R}$ symbolizes the sample recording process using the target offset t_i.

$$k_i = \mathcal{C}(\mathcal{S}_{t_i}) \tag{5.4}$$

We realized the trigger oracle by using the victim mbedTLS implementation and extending the API with the possibility to use a user-defined key. We build the oracle enclave with debug mode enabled and use the profiling functionality as we describe in Section 4.7 to record the next *start*-trigger. We then use this recorded *start*-trigger and attack the victim implementation, gathering samples of either the *one* case or the *zero* case.

We configured the *measurement framework* to use the *asymmetric* trigger-type since we start the measurement when we hit the start trigger and stop it after we recorded two

```
reconstruct_key() → K_{N_bit}:
    k_0 = 1
    for i in (1, ..., N_bit):
        t_i = O(i, (k_0, ..., k_{i-1}), M, N, L)
        S_{t_i} = R(t_i)
        k_i = C(S_{t_i})
```

Listing 5.7: Iterative reconstruction of the victim key.

instructions. The single-step trigger was set to 255, and the measurement trigger was set to 511, so we measure exactly two instructions. We replaced the complex AEP callback with the `and` version (see Section 4.4) to reduce the measurement noise. We execute a `hlt` delay of 5000 cycles before each measurement to bring the CPU to a known state before we *zero*-step the instructions.

Results Figure 5.5 shows the result of the core-voltage domain of the attack. We record ten traces per key bit and plot the distribution of the samples as candlesticks. We can observe that the core-voltage domain has different distributions depending on the instruction being single stepped. For the AVX instruction, i.e., the one bit, we can see a nearly punctual distribution. For the zero bit, we can see a broad spread of the core voltage distribution.

$$k_i = \mathcal{C}(\mathcal{S}_{t_i}) = (min_{V_{core}}(\mathcal{S}_{t_i}) > 1.0025) \tag{5.5}$$

To classify the secret key, we use a minimum classifier and a threshold as Equation 5.5 shows. The complete measurement took under 5 minutes and managed to reconstruct all 71 bits of the private key. The public modulus N had a length of 512-bit. We used the private key $D =$ `0x660755fff0f0555537`.

Chapter 6

Countermeasures

In this chapter, we discuss how to mitigate the attacks we show in Chapter 5. We discuss countermeasures against classical power analysis attacks in Section 6.1. We then propose our countermeasures to mitigate the RAPL and core-voltage side channels in Section 6.2.

6.1 Power Analysis Countermeasures

In this section, we discuss common countermeasures against classical power analysis attacks.

Hiding is a mitigation technique where the power leakage of an operation containing the secret information is hidden from the attacker by mixing irrelevant operations into the overall power trace [4]. Since we assume known assembly attacks and target specific instructions with our framework, software-based hiding does not help against our *zero*- and *single*-stepping attacks.

One of the countermeasures to harden an RSA decryption is exponent blinding [4]. We discussed the basic principle of exponent hiding in Section 2.6. The idea is to change the exponent of the *square-and-multiply* algorithm with every decryption. Therefore, the attacker cannot see the same key bits over multiple traces. This increases the duration of the attack, as Schindler et al. [46] showed. Exponent blinding does not add security if the attacker is capable of single-trace attacks, as we discuss in Section 7.2.

Exponent blinding is a form of masking. Masking combines the secret information with a random mask to make the result of the operation independent of the secret value [4]. Masking is a form of secret sharing where the information is split up into multiple shares representing the real intermediate. Exponent blinding is a particular case where the masking does not have to be undone after the mask is applied, due to the properties of RSA (see Section 2.6).

6.2 RAPL and Core-Voltage Countermeasures

To mitigate power side-channel attacks using the RAPL interface or the core-voltage side channel, we propose to either prohibit energy readings from userspace or to introduce a mechanism like the *perf* interface paranoia [63]. The paranoia level can be set by the root user to allow readings from unprivileged users, which are disabled by default.

For the SGX threat model, where the attacker has kernel-level privileges, we propose not to include the energy consumption of instructions executed inside an SGX enclave in the energy readings and fix the core-voltage field to the value during the entry of the enclave, during Asynchronous Enclave Exits (AEX) the core voltage must be reverted to the value during the enclave entry and enforced that consecutive changes do not leak information about the SGX instructions. If the energy consumption of SGX is not included in the RAPL measurements, energy-aware applications will no longer be accurate. Therefore, we propose to add a fixed energy increment per time to the RAPL domains based on the current P-state. These mitigations can either be deployed by a microcode update or might need hardware changes.

Decreasing the resolution of the RAPL interface does not mitigate software-based power side channels since we still can simply increase the measurement duration of our *zero*- and *single*-stepping attack (see Section 5.4). However, decreasing the resolution might make some attacks infeasible.

Chapter 7

Limitations and Future Work

In this section, we discuss ideas and future work arising from this thesis. We classified the different types of victim algorithms in Section 4.6. Since we covered attacks on *symmetric* and *asymmetric* algorithms in Chapter 5, we discuss in Section 7.1 how to extend the attack on *constant* algorithms.

Beyond the attacks described in this work, we also mounted further attacks as described in our paper by Lipp et al. [16], which is currently in submission to the USENIX Security 2021 conference. The attacks in this thesis and additional attacks from the paper were disclosed to Intel, and the attacks received two CVE numbers (CVE-2020-8694 and CVE-2020-8695). Lipp et al. [16] discuss possible userspace attacks and additionally demonstrate an attack on `AES-NI`.

7.1 Attack on a Constant Algorithm

As we described in Section 4.6, *constant* algorithms use the secret information in the data operands of instructions. Therefore, cryptographic constant time algorithms (see Section 2.6) fall into this category.

We looked at the implementation of bearSSL [48], a constant time library, providing a constant time implementation of the *square-and-multiply* algorithm, similar to the version we describe in Listing 2.2. The algorithm uses a constant time `memcpy` function inside the loop iterating over the private key length. This `memcpy` function lies on a different page than the loop, and we can, therefore, trigger a page fault for every distinct key bit. This enables us to target instruction even more precisely than with the normal precise single stepping because stepping errors will not add up. In addition, we can only use targeted *single*-stepping mechanisms after we observed the page fault. Therefore, we do not need to precisely single step over the instructions before the constant time `memcpy` function is called, increasing the attack performance dramatically.

Listing 7.1 shows the assembly code of the constant time `memcpy` function. The key

```
1      br_ccopy:
2          xor     %r8d,%r8d
3  ->      neg     %edi
4      br_ccopy_loop:
5          cmp     %r8,%rcx
6          je      br_ccopy_end
7          movzbl  (%rsi,%r8,1),%r9d
8          mov     (%rdx,%r8,1),%al
9          xor     %r9d,%eax
10         movzbl  %al,%eax
11 ->      and     %edi,%eax
12 ->      xor     %r9d,%eax
13         mov     %al,(%rsi,%r8,1)
14         inc     %r8
15         jmp     br_ccopy_loop
16     br_ccopy_end:
17         retq
```

Listing 7.1: Target instructions containing the secret information.

bit information gets passed to the function in the `edi` register. The instructions marked with an arrow are the instructions containing the secret information. We can target each of these instructions with the targeted *zero*- and *single*-stepping mechanism to record power side-channel information.

As Lipp et al. [16] show, these different operands in the `and` instruction have different energy signatures and can be detected with the RAPL interface. Since the operands of the `and` and `xor` operation are either the value zero or the data to copy, we should be able to distinguish the energy signature. The number of traces recorded must be way higher in comparison to the toy example from Section 5.4. Also, the measurement noise must be reduced, and therefore, the low-level assembly handler might be used. Due to time limitations, this experiment was not completed for this thesis.

7.2 Single Trace Attack

Our attack on mbedTLS from Section 5.5 can be extended to a single trace attack. Currently, we record ten samples per target instruction and then apply the classification to determine the next target instruction. We used ten different traces to record information to reduce potential noise. We could instead use a single trace of the victim and simply record measurement samples by only *zero*-stepping the target instruction.

We then could use the recorded samples inside the AEP callback to classify the next target instruction. This would enable dramatically faster attacks since the longest phase of the attack is to single-step to the target instruction. With single-trace attacks, we also can attack implementations that apply exponent blinding in the decryption method because we can observe every bit of the blinded exponent and, therefore, reconstruct

private keys for which Equation 2.11 holds.

7.3 Increasing the Execution Time

In our classical SPA attack in Section 5.3, we observe limitations of the key bit resolution we can reconstruct. To slow down the decryption of the victim, i.e., increase the time per key bit, we could change the type of the data pages of the enclave to use uncacheable memory. This would slow down each memory load and store of the enclave and potentially increase the resolution to reconstruct distinct key bits. Disabling all the caches with the *Cache Disable* bit in the `cr0` [5] register is not applicable because we observed instabilities with the measurement framework.

7.4 Other (Micro)-Architectures

This thesis focused only on Intel CPUs, but different manufacturers also provide software-based energy readings.

AMD Sine the ZEN CPU generation AMD also provides a RAPL interface [64]. In contrast to the RAPL interface from Intel, the AMD version does not include all the cores in the *core* domain, instead, each of the cores provides its own per core register. This eliminates all noise from the other cores.

Currently, the register can only be read from the kernel. Since AMD does not provide a feature like SGX the threat models to attack userspace applications do not include kernel-level privileges. However, the AMD RAPL interface will soon also be supported from userspace [30], allowing unprivileged reads. This might enable power side-channel attacks similar to the ones described by Lipp et al. [16] and this thesis

ARM ARM does not provide a direct software interface like RAPL. However, some ARM boards like the *ARM CoreTile Express A15x2* [65] provide an energy meter built onto the development board. The energy meter provides a sample rate of 10 kHz and also accumulates the energy in the microjoule range. Therefore, the resolution is comparable to that of the *package* and the *core* RAPL domains. Hence, this energy meter can potentially replace the RAPL interface. Vasilakis et al. [66] used a similar board to characterize the energy consumption of different ARM instructions.

Chapter 8

Conclusion

In this master thesis, we presented a framework to record power side channel relevant information on instruction-level granularity. We refined the capabilities of the RAPL interface and the general idea that the interface leaks security-relevant information, even if the code is protected by an SGX enclave. We extended power side-channel attacks by exploiting a voltage-based side channel purely from software.

With the help of the local APIC timer, we developed a measurement framework to record power side-channel information and used the `hlt` instruction to extend the RAPL interface resolution. With the extended measurement resolution, we reconstructed the underlying moving average signal. We applied a classical SPA attack on the mbedTLS library, which was able to reconstruct consecutive bit blocks of size eight.

We enhanced the attack capabilities even further by adopting a precise execution control method to record side-channel information on a per instruction-level granularity. With this extension, we showed that we can distinguish two different branches within the same cache line and were able to reconstruct the secret key. We mounted an attack on the mbedTLS library and showed that these techniques are also applicable to large scale libraries, with our precise instruction targeting technique. We managed to leak a 71-bit long RSA key with a 512-bit long public modulus in under five minutes with ten traces per key bit.

We disclosed our findings with collaboration with Lipp et al. [16] to Intel and received two CVE numbers for these types of attacks.

We conclude that the general idea to support power-aware computing by implementing a software-based energy monitoring facility conflicts with the requirements to implement a side-channel-secure system. We proposed mitigations to allow the coexistence of these different requirements by removing power-related information that originates from SGX enclaves. In summary, this thesis closes the gap between hardware and software-based power side-channel attacks.

Bibliography

[1] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *CRYPTO'99*, 1999.

[2] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, 2011.

[3] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *International workshop on cryptographic hardware and embedded systems*. Springer, 2004, pp. 16–29.

[4] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks: Revealing the secrets of smart cards*. Springer Science & Business Media, 2008, vol. 31.

[5] Intel®, "Intel® 64 and IA-32 Architectures Software Developer′s Manual, Volume 3 (3A, 3B & 3C): System Programming Guide," no. 325384, 2016.

[6] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using rapl," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 3, pp. 13–17, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2425248.2425252

[7] H. Mantel, J. Schickel, A. Weber, and F. Weber, "How secure is green it? the case of software-based energy side channels," in *European Symposium on Research in Computer Security*. Springer, 2018, pp. 218–239.

[8] M. M. Fusi, "Information-leakage analysis based on hardware performance counters." [Online]. Available: https://www.politesi.polimi.it/bitstream/10589/137507/3/2017_12_FUSI.pdf

[9] Intel®, "Intel® software guard extension." [Online]. Available: https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html

[10] M. Schwarz and D. Gruss, "How trusted execution environments fuel research on microarchitectural attacks," *IEEE Security & Privacy*, 2020.

[11] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 640–656.

[12] J. Van Bulck, F. Piessens, and R. Strackx, "Sgx-step: A practical attack framework for precise enclave execution control," in *Workshop on System Software for Trusted Execution*, 2017.

[13] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic," in *CCS*, 2018.

[14] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against intel sgx," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[15] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.

[16] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "With great power comes great leakage: Exploiting software-based power side channels on x86," in *USENIX Security*, 2021.

[17] J. Aumasson and L. Merino, "Sgx secure enclaves in practice," *Blackhat*, 2016.

[18] F. Wanlass and C. Sah, "Nanowatt logic using field-effect metal-oxide semiconductor triodes," in *1963 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, vol. 6. IEEE, 1963, pp. 32–33.

[19] H. Weste Neil and H. David, "Cmos vlsi design: a circuits and systems perspective," 2006.

[20] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *S&P*, 2019.

[21] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security*, 2018.

[22] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "NetSpectre: Read Arbitrary Memory over Network," *arXiv:1807.10535*, 2018.

[23] J. D. Golić and C. Tymen, "Multiplicative masking and power analysis of aes," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2002, pp. 198–212.

[24] D. Genkin, A. Shamir, and E. Tromer, "Rsa key extraction via low-bandwidth acoustic cryptanalysis," in *Annual Cryptology Conference*. Springer, 2014, pp. 444–461.

[25] J. Haj-Yahya, A. Mendelson, Y. B. Asher, and A. Chattopadhyay, *Energy Efficient High Performance Processors: Recent Approaches for Designing Green High Performance Computing*. Springer, 2018.

[26] The Linux Kernel, "intel_pstate." [Online]. Available: https://www.kernel.org/doc/html/v4.12/admin-guide/pm/intel_pstate.html

[27] Archlinux Wiki, "cpufreq." [Online]. Available: https://wiki.archlinux.org/index.php/CPU_frequency_scaling

[28] Intel®, "Intel® rapl power meter." [Online]. Available: https://01.org/rapl-power-meter

[29] Linux Kerenl Documentation, "Linux powercap documentation," 2019? [Online]. Available: https://www.kernel.org/doc/Documentation/power/powercap/powercap.txt

[30] M. Larabel, "Amd zen/zen2 rapl support merged in linux 5.8," 2020. [Online]. Available: https://www.phoronix.com/scan.php?page=news_item&px=AMD-Zen-RAPL-Linux-5.8

[31] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan, "Power-management architecture of the intel microarchitecture code-named sandy bridge," *Ieee micro*, vol. 32, no. 2, pp. 20–27, 2012.

[32] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "Containerleaks: Emerging security threats of information leakages in container clouds," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 237–248.

[33] M. Wolf, *High-performance embedded computing: applications in cyber-physical systems and mobile computing*. Newnes, 2014.

[34] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "Voltjockey: Breaking sgx by software-controlled voltage-induced hardware faults," in *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 2019, pp. 1–6.

[35] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, "V0ltpwn: Attacking x86 processor integrity from software," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[36] E. Rescorla and T. Dierks, "The transport layer security (tls) protocol version 1.3," 2018.

[37] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," 1999.

[38] A. Bauer, E. Jaulmes, E. Prouff, and J. Wild, "Horizontal and vertical side-channel attacks against secure rsa implementations," in *Cryptographers' Track at the RSA Conference*. Springer, 2013, pp. 1–17.

[39] S.-M. Yen and M. Joye, "Checking before output may not be enough against fault-based cryptanalysis," *IEEE Transactions on computers*, vol. 49, no. 9, pp. 967–970, 2000.

[40] OpenSSL, "Openssl: The open source toolkit for ssl/tls." [Online]. Available: http://www.openssl.org

[41] Paul Mutton, "Certificate revocation: Why browsers remain affected by Heartbleed," 2014. [Online]. Available: https://news.netcraft.com/archives/2014/04/24/certificate-revocation-why-browsers-remain-affected-by-heartbleed.html

[42] ARMmbed, "Mbed TLS ," 2019, retrieved on April 8, 2019. [Online]. Available: https://tls.mbed.org/

[43] R. L. Rivest, A. Shamir, and L. M. Adleman, "Cryptographic communications system and method," 1977, uS Patent Grant 1983-09-20. [Online]. Available: https://patents.google.com/patent/US4405829

[44] J. Katz, A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 1996.

[45] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P*, 2015.

[46] W. Schindler, "Exclusive exponent blinding may not suffice to prevent timing attacks on rsa," in *International Workshop on Cryptographic Hardware and Embedded Systems*, 2015.

[47] Intel®, "Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations." [Online]. Available: https://software.intel.com/security-software-guidance/insights/guidelines-mitigating-timing-side-channels-against-cryptographic-implementations/

[48] T. Pornin, "Bear ssl." [Online]. Available: https://bearssl.org/

[49] A. S. Tanenbaum and H. Bos, *Modern operating systems*. Pearson, 2015.

[50] J.-P. Aumasson and L. Merino, "SGX Secure Enclaves in Practice: Security and Crypto Review," in *Black Hat Briefings*, 2016.

[51] A. Adamski, "Overview of intel sgx." [Online]. Available: https://blog.quarkslab.com/overview-of-intel-sgx-part-1-sgx-internals.html

[52] V. Costan and S. Devadas, "Intel SGX explained," 2016.

[53] S. Gueron, "A memory encryption engine suitable for general purpose processors," ePrint 2016/204, 2016.

[54] Intel®, "Intel® sgx: Debug, production, pre-release –what's the difference?" [Online]. Available: https://software.intel.com/content/www/us/en/develop/blogs/intel-sgx-debug-production-prelease-whats-the-difference.html

[55] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1041–1056.

[56] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 88–105.

[57] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 3–24.

[58] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann, 2017.

[59] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," *arXiv:1811.05441*, 2018.

[60] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar *et al.*, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 769–784.

[61] M. Schwarz, "Pteditor." [Online]. Available: https://github.com/misc0110/PTEditor

[62] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "Microscope: Enabling microarchitectural replay attacks," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. ACM, 2019, pp. 318–331.

[63] L. P. Manual, "perf_event_open." [Online]. Available: https://man7.org/linux/man-pages/man2/perf_event_open.2.html

[64] AMD, "Register referencefor amd family 17h processors." [Online]. Available: https://developer.amd.com/wp-content/resources/56255_3_03.PDF

[65] ARM, "Arm coretile express a15x2 technical reference." [Online]. Available: https://developer.arm.com/documentation/dui0604/f/

[66] E. Vasilakis, "An instruction level energy characterization of arm processors," *Foundation of Research and Technology Hellas, Inst. of Computer Science, Tech. Rep. FORTH-ICS/TR-450*, 2015.