Karlsruhe Institute of Technology

Covert Channel based on AMD Precision Boost 2

Bachelor's Thesis
submitted by

Tim Schmidt

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Mathias Gottschlag M.Sc.

27. Juni 2019 – 28. Oktober 2019

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, October 28, 2019

Abstract

Covert channels provide an attacker with the means of bypassing application isolation demanded by system security policies. This thesis presents a frequency-based covert channel using the dynamic frequency scaling technology AMD Precision Boost 2, similar to a covert channel based on Intel Turbo Boost presented by Kalmbach [8]. By applying load on multiple CPU cores, the core frequency of all cores is reduced. This frequency drop can be measured by a receiver, allowing messages to be transmitted. Our analysis showed that in contrast to Intel Turbo Boost, Precision Boost 2 reacts with high, asymmetric latencies which introduce new problems in the construction of covert channels. In our design, we compensate for the asymmetric latencies by changing how the receiver translates frequencies into symbols. Our covert channel reaches a net bit rate of 1.08 bit/s when using a transfer protocol to ensure that messages are transmitted without errors. The channel also does not rely on operating system support, making the construction of centralized software-based countermeasures not trivial.

Contents

Chapter 1

Introduction

Applications often come into contact with sensitive information. To reduce the risk of this information being leaked, system security policies can demand that applications processing sensitive data be isolated from the internet and other applications. This way, even if an attacker manages to take control of an application processing sensitive data, he is not able to exfiltrate the data. Covert channels, however, provide the means to bypass this security measure. By utilizing the side-effects that certain instructions or instruction sequences have on shared resources such as the CPU, two processes are able to communicate without using standard communication channels monitored by the operating system.

A new kind of covert channel utilizing Intel's dynamic frequency scaling technology called Turbo Boost was recently presented by Kalmbach [8]. His covert channel works by applying load on multiple CPU cores, causing the core frequency of all cores to drop. This frequency drop can be measured, allowing information to be transmitted. Kalmbach's covert channel was only tested with Intel CPUs and relies on specific traits of Intel Turbo Boost. This thesis presents a covert channel based on AMD's counterpart called Precision Boost 2. We analyze the behaviour of Precision Boost 2 by monitoring frequency changes caused by a change in processor load. During our analysis, we found key differences between Precision Boost 2 und Turbo Boost in the way that they regulate frequency. While Turbo Boost considers the amount of active cores and updates the frequency every millisecond, Precision Boost 2 "only assesses whether the processor is within specifications" [6], according to publications by AMD. We found that Precision Boost 2 reacts slower to changes in processor load and produces asymmetric latencies for rising and falling frequencies. This behaviour is accounted for in the construction of the covert channel by implementing latency correction. Our covert channel reaches a net bit rate of 1.08 bit/s, more than what is considered "acceptable in most application environments" in the "Orange book" [13]. In addition to the covert channel, we present an approach on how the same technique could

be used to mount a side channel attack on third-party applications running on the same system, allowing the attacker to gain information on when the application produced load. Just as Kalmbach's covert channel, this covert channel does not rely on operating system support, making the construction of centralized software-based countermeasures not trivial.

We start our thesis by explaining some of the technologies and techniques on which the covert channel relies in Chapter 2 and then conduct an extensive analysis into the behaviour of Precision Boost 2 in Chapter 3. Afterwards, we use the results of the analysis to design and implement a covert channel in Chapter 4 and evaluate the performance of a prototype in Chapter 5. Finally, we draw a conclusion and discuss future work in Chapter 6.

Chapter 2

Background

In this section, we provide background information on technologies and techniques on which the covert channel presented in this thesis relies. In Section 2.1, we first describe side channels and covert channels in general as well as provide examples of previously discovered side channels. We then present an attack scenario that utilizes the covert channel presented in this paper and provide information on the used hardware and the AMD Precision Boost 2 technology. The basics of the Transfer Control Protocol (TCP) are also described, as a simplified version of TCP is used to build a reliable channel on top of the covert channel.

2.1 Side Channel & Covert Channel

A lot of operations performed on computers have measurable side effects, such as increased power usage when executing AVX instructions [10], noise produced by the moving head of mechanical hard drives or state changes in microarchitectural components [11]. Any kind of side effect that an attacker can observe and use to obtain information about the operation that was performed or the state that a machine is in is described as a *side channel*.

Side channels have been exploited since at least as early as 1943 [3], when sound waves and radiation emitted by switches and contacts in then-used machines could be measured from half a mile away. Modern side channel attacks, for example, use the size of HTTPS requests to deduce information about the content. This is especially effective when the content is known to have a specific structure.

A special case of side channels are *covert channels*. Covert channels are side channels that are used deliberately by two actors, called sender and receiver, to transmit data between them. As these channels are usually not monitored by the operating system, they can be used to establish communication even when the two

actors are supposed to be isolated from one another according to system security policies.

In this thesis, the change in CPU frequency caused by high load in combination with dynamic frequency scaling technologies such as AMD Precision Boost 2 is used to establish communition between two processes without the need for operating system support.

The attack scenario used throughout this thesis consists of two processes. While the sender process has access to sensitive data, it is not connected to the internet and is not allowed to communicate with other processes according to the system security policy. The receiver process is connected to the internet. Both processes are controlled by the attacker, though the attacker does not have any way to interact with the sender once it aquires the sensitive data. To exfiltrate the sensitive data, the attacker must establish communication between the sender and the receiver.

The existence of covert channels is considered a security risk. The severity of this security risk is measured by the bandwidth of the covert channel and the performance loss incurred when implementing measures to reduce the bandwidth. The U.S. Department of Defense considers covert channels with bandwidths below 1 bit/s "acceptable in most application environments" [13]. While the covert channel presented in this thesis only reaches 1.08 bit/s in its current form, several hardware-based covert channels with very high bandwidths are known as well. One such variant is a cache-based covert channel called *Flush+Reload* that was first presented by Yarom et al. [16] in 2014. This covert channel has been measured to support sending 298 kB/s with an error ratio $< 0.005\,\%$ [4].

Covert channels have been constructed with a plethora of transmission media, including the timing of IP packets as described by Cabuk et al. [2] and electromagnetic waves emitted by a USB data bus as described by Guri et al. [5]. While the covert channel described in this thesis relies on the modification of CPU frequencies, it is not the first covert channel of its kind. Another covert channel which exploits dynamic voltage and frequency scaling (DVFS) was presented by Miedl et al. [12] in 2018 and a covert channel similar to the one presented in this thesis was described by Kalmbach [8] in 2019. Kalmbach's covert channel is based on Intel Turbo Boost. While Intel Turbo Boost offers comparatively low-latency frequency changes when the amount of active CPU cores change, the dynamic frequency scaling technology examined in this paper, AMD Precision Boost 2, reacts with high, asymmetric latencies which introduce new problems in the construction of covert channels. These high latencies force the sender to wait for the frequency to stabilize before sending a new symbol, reducing the symbol rate. The asymmetry of the latencies causes the receiver to always read one kind of symbol for longer than intended by the sender. With symmetrical latencies, the effects of the latencies on the length of symbols cancel each other out. In our

case, however, we have to compensate for latency before translating frequencies into symbols. Another similar covert channel also released in 2019 by Khatamifard et al. [9] works by changing the power usage of the sender to affect the power budget share, called power headroom, of the receiver core and thus change the core frequency in a way that can be measured by the receiver. While this thesis also briefly considers power usage to support speculations made during the analysis in Chapter 3, the covert channel presented here communicates entirely via changes in the CPU frequency caused by Precision Boost 2.

2.2 Architecture

The experiments made in this thesis are run on a system equipped with an AMD Ryzen 7 2700 processor using the AMD Zen+ microarchitecture and microcode version `0x800820b`. Preliminary tests are also made using an AMD Ryzen 7 3700X processor using microcode version `0x8701011`.

2.2.1 AMD Precision Boost 2

While both processors implement dynamic frequency scaling, they are based on different microarchitectures. The Ryzen 7 2700 uses the Zen+ microachitecture while the Ryzen 7 3700X is based on Zen2. The Zen+ microarchitecture was presented by AMD in 2018 and supports, among other things, higher base and turbo frequencies compared to its predecessor [15]. The dynamic frequency scaling technology used in the Zen+ microarchitecture is called AMD Precision Boost 2. It is referred to as Precision Boost 2 from now on.

Whenever processor cores are idle, the CPU uses less power and produces less heat than allowed by its *Thermal Design Power* (TDP). The TDP is the maximum amount of heat generated by the processor that can reliably be dissipated by the cooling system.

If a CPU is running below its TDP because of idle cores while other cores are at full load, it is effectively wasting time. To combat this, dynamic frequency scaling technologies like Precision Boost 2 have been developed that increase the frequency of individual cores whenever enough other cores are idle.

As described on the AMD community platform by AMD employee Robert Hallock [6], Precision Boost 2 can increase the core frequency past the base frequency in 25 MHz steps until a set maximum turbo frequency is reached or the TDP is exhausted. Precision Boost 2's predecessor, Precision Boost 1, has two different modes that determine the maximum turbo boost frequency: *Two-core boost* and *all-core boost*. Two-core boost is activated when at most two cores have load and offers a greater maximum frequency. All-core boost is activated as

soon as any core goes idle. According to rhallock [6], Precision Boost 2 no longer enforces a lower maximum turbo frequency when more than two cores are active. An analysis of the behaviour of Precision Boost 2 under different load patterns is conducted in Chapter 3.

2.2.2 Core Complex

AMD processors with the Zen+ microachitecture feature modules called Core Complex (CCX) that each contain four processor cores and their L1, L2, and L3 CPU caches [7]. The 8-core Ryzen 7 2700 includes two such CCXs that are connected using AMDs interconnect architecture *Infinity Fabric*. This seperation of cores into two groups is suspected to be the cause of some of the findings in Chapter 3.

2.2.3 Difference to Intel Turbo Boost

A similar technology, called Intel Turbo Boost, is used on modern Intel CPUs and activates when a core reaches the C3 sleep state. Intel Turbo Boost is documented to update the core frequencies every millisecond. A very similar covert channel based on Intel Turbo Boost presented by Kalmbach [8] manages to reach the theoretical maximum bandwidth of 1000 bit/s under specialized circumstances with an error ratio of 13.5 %. Under more realistic circumstances and when using a simple transfer protocol similar to TCP to ensure the reliability of the communication channel, a bandwidth of 56 bit/s is achieved.

2.3 Transmission Control Protocol

In order to combat transmission errors introduced by scheduler decisions and noise generated by fluctuating decisions of Precision Boost 2, a simplified version of the Transmission Control Protocol (TCP) is used to produce a reliable communication channel. TCP splits the user data into several packets that each contain a sequence number and a checksum. Upon receiving a TCP packet, the receiver can check the integrity by calculating the checksum of the received user data and comparing it to the checksum contained in the packet. The checksum algorithm is chosen such that the probability of falsely identifying a packet as uncorrupted is negligible.

The sender acknowledges each successfully received package by sending an acknowledgement packet (ACK) containing the sequence number of the received packet. This allows the sender to detect whether a packet was transmitted successfully or has to be sent again.

The protocol used in this thesis is a simplified version of TCP, lacking advanced features such as port numbers, control bits, and congestion control.

2.4 Running Average Power Limit

Running Average Power Limit (RAPL) is an interface designed by Intel with which a system administrator can read and limit the power usage of CPU packages and DRAM [14]. Power can be measured by reading the RAPL counters, a set of model-specific registers that are also found on Ryzen CPUs [1]. When using the Linux Kernel 5.1.16, the kernel module `msr` must be loaded to access these registers. As this module is not loaded by default and loading it requires operating system support, they are not suitable for use in a covert channel that is supposed to work without operating system support. We do, nevertheless, use power readings obtained through the RAPL interface to support some of the speculations made during the analysis in Chapter 3.

Chapter 3

Analysis

Before we can construct a covert channel based on AMD Precision Boost 2, the exact behaviour when adding or removing load from CPU cores has to be known. To achieve this, an extensive analysis of the concrete parameters of Precision Boost 2 is conducted in this section. Afterwards, we also consider whether a side channel can be built based on Precision Boost 2.

The goal of this analysis is to find out how strong and how fast the frequency on cores with high load changes in reaction to other cores becoming active or idle. In order to measure this, one core is designated to be the *measurement core* while all others are designated *load cores*. On the measurement core, the core frequency and power usage are measured by a *measurement process* while the load cores are each assigned one *load process* that can either run at full load or sleep using the `usleep` function.

3.1 Methodology

As this thesis concerns itself with the creation of a covert channel, the goal of the measurements in this section is to discover the maximal values for the following two properties:

- **Symbol rate.** How often can the load be changed without making frequency changes unrealiable? Figure 3.1 shows the limiting effects that the latency of frequency changes can have on the maximum symbol rate.
- **Amount of possible symbols**. How many different frequency levels are there that the receiver can distinguish reliably? The covert channel is going to use bits as units of information. A very simple translation of symbols to bits is possible when the amount of symbols is a power of two. E.g. a set of 16 different possible symbols would equal 4 bits of information per symbol.

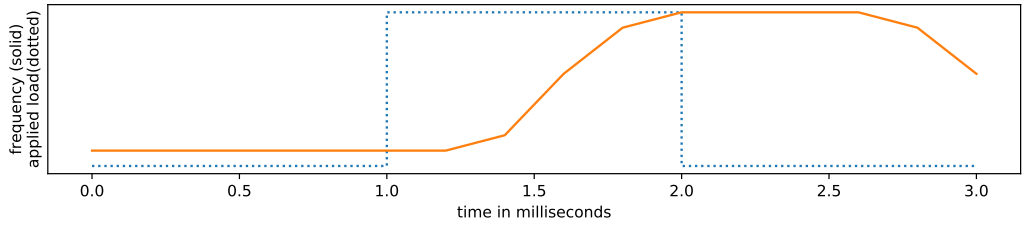


Figure 3.1: Possible frequency change when applying a *low-high-low* load pattern. A high latency limits the maximum symbol rate because the frequency needs time to adapt to the new load. In this example, setting to the load to *high* at the 3 ms mark would cause the frequency to rise before hitting its mininum, possibly hiding the low-load symbol sent inbetween.

To determine the maximum symbol rate and symbol amount, we measure the latency with which the frequency on the measurement core updates when the load changes. While the load on the measurement core remains the same throughout the analysis, the load on the load cores can be changed in multiple ways:

- **Load configuration.** The processor used in this thesis has eight physical cores. One of these will be used by the measurement process, leaving seven load cores. By selectively sending load processes to sleep or waking them up, we can change the set of active processor cores. This set will be called *load configuration* from now on.
- **Load instructions.** We can change the instructions that the load processes use during high load periods. Candidates include `NOP` and AVX instructions.
- **Load pattern.** We can choose the set of load configurations that are used during a measurement run and vary their order. Possible orders are:
  - 0 load cores active $\rightarrow$ 8 load cores active $\rightarrow$ 0 load cores active
  - 0 load cores active $\rightarrow$ 1 load core active $\rightarrow$ 2 load cores active ...
- **Interval length.** We can vary the amount of time that one load configuration stays active.

The measurement process is supposed to observe the core frequency. To measure core frequency, the measurement process repeatedly increases a counter in a short loop, called the *spin counter*. How far the counter can be increased in a set time period can then be used as a measure of core frequency. As the loop increasing the counter value spans multiple instructions and is influenced by scheduler

decisions that pause the measurement process, the counter value is not an accurate measurement of core frequency in hertz. The spin counter does, however, scale linearily with the core frequency, so it can be used to compare multiple frequencies. As frequency measurements using a spin counter do not require operating system support, the same technique can be used later on when constructing the covert channel.

In addition to the value of the spin counter, the latency with which these values react to changes on the load cores is also of importance, as slow reactions can affect the covert channel bandwidth negatively. To accurately determine the latency of frequency changes, the frequency is measured every 0.1 ms.

3.1.1 Setup

The system used in this section and throughout the rest of this thesis contains an AMD Ryzen 7 2700 processor and runs the Arch Linux operating system with minimal packages and background services. The used Linux kernel is at version 5.1.16.

During the experiments, we apply load to specific cores. To decrease the chance of the scheduler interfering with our core assignment, all but one cores are isolated with the `isolcpus` kernel parameter. This prevents the scheduler from moving processes to or from the isolated cores. At least one core has to remain non-isolated for the scheduler to work. In our analysis, the non-isolated core is the measurement core. Processes can still be assigned to the isolated cores by using the `sched_setaffinity` function. To make sure the frequency governor actually tries to run the cores at maximum frequency, the default `schedutil` governor is replaced by the `performance` governor. To simplify the analysis, simultaneous multithreading (SMT) is disabled.

3.2 Measurements

Our measurement process is set to always run on core 0. The load cores 1-7 are isolated using `isolcpus`. This means that the measurement process shares its core with all remaining processes running on the system. Apart from one load process per load core, no processes are assigned to the load cores, so that they do not wake up unintentionally. The load configurations in the following diagrams are represented as a sequence of eight numbers in which a `1` represents an active core and a `0` represents an inactive core. Because the measurement process will always keep core 0 active, all load configurations start with a `1` and the sequence `1-0-0-0-0-0-0-0` represents a load configuration in which all load cores are idle. Two load configurations are said to have the same load when they

have an equal amount of active load cores, for example `1-1-1-0-0-0-0-0` and `1-1-0-0-1-0-0-0` both have two active load cores.

Different load configurations for the same test are always used in sequence, not simultaneously, with cooldown and warmup phases between the measurements to insure that the results are independent of one another. In the cooldown phase, no load cores are active. In the warmup phase, the load configuration that is to be used in the measurement is active. Both the cooldown and warmup phase lasts long enough for the measurement core to reach a resting core frequency.

Note that the load instructions used to create load during these measurements is a sequence of AVX256 instructions. The usage of different types of load instructions is discussed in Section 3.2.6.

3.2.1 Test 1: Constant load

This first test examines the resting frequencies of different load configurations. The results are displayed in Figure 3.2. A common occurence with all load configurations are short-term frequency drops. We presume that these occur when the measurement process is paused by the scheduler, resulting in less spins counted during the affected 0.1 ms counting interval.

The figure also shows a larger gap between the frequencies observed with the load configurations `1-1-1-1-0-0-0-0` and `1-1-1-1-1-0-0-0`. This might be caused by the fact that the processor cores on the Ryzen 7 2700 are seperated into two modules called *Core Complex* (CCX). The first load configuration uses only one CCX while the second one has active cores on both CCXs. We assume that this increases the chip's power usage and causes Precision Boost 2 to lower the frequency multiplier.

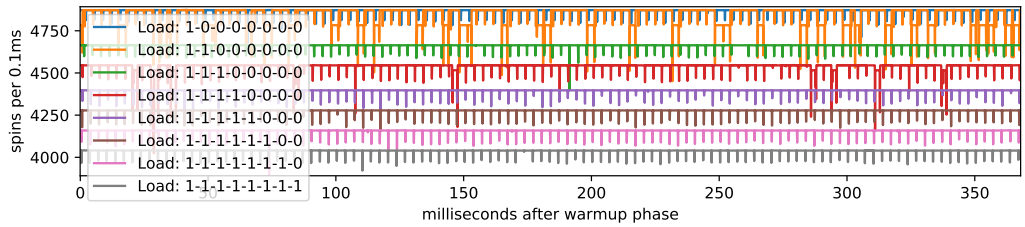


Figure 3.2: Core frequency under different load configurations. While enabling a single load core does not decrease the resting core frequency, it causes short-term frequency drops to happen more often.

To increase readability, small frequency drops are filtered out in the following diagrams using a sliding window median filter with a 1 ms window size. Figure 3.3 shows the same test with other load configurations. Slight differences can

be observed when the same load is distributed across different Core Complexes. Most of the frequency drops still seen after applying the median filter end up at the same frequency, suggesting that the these drops are caused by Precision Boost 2 lowering the turbo frequency multiplier. As the frequency multiplier is documented to work in 25 MHz steps [6], the core frequency can only jump to a few discrete frequencies. Since the differences between the different load configurations with the same load are very small and no frequencies between their levels can be observed, they are probably only one or two multiplier steps apart. This makes any attempt of distinguishing two of these configurations very susceptible to noise.

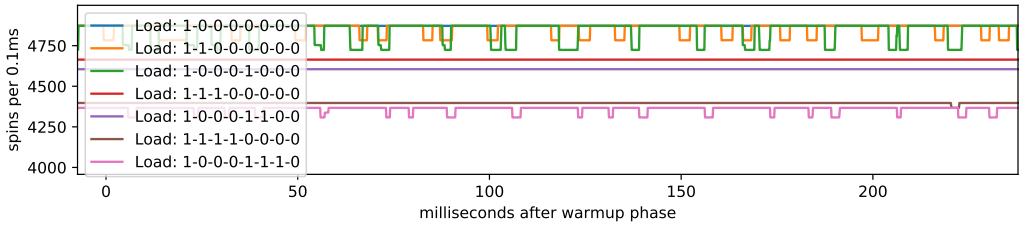


Figure 3.3: Core frequency under different load configurations. Slight differences can be seen when distributing the same load onto a different set of load cores.

3.2.2 Test 2: Rising load

The following few tests measure the frequency change when increasing load over time. Figure 3.4 shows how the frequency changes when load is increased from zero load cores to a specific load configuration without any intermediate steps. As the graph shows, the frequency starts dropping after 0.5 ms to 3.5 ms and reaches a resting level after 5.7 ms to 7.9 ms. This suggests that, in some situations, Precision Boost 2 reacts faster than Intel Turbo Boost, which is documented to update the frequency in 1 ms intervals. There is, however, a longer latency before the frequency reaches its resting level.

Figure 3.5 shows that switching between different load configurations sometimes causes Precision Boost 2 to reduce the frequency multiplier further than when switching between a single load configuration and no load. For comparison, the second load level induces a frequency of 4575 spins in the first half and a frequency of 4515 in the second half of Figure 3.5. The third level reaches a resting frequency in the first half while showing heavy oscillations in the second half. This instability poses a challenge when trying to use multiple load configurations to increase the amount of possible symbols for the covert channel.

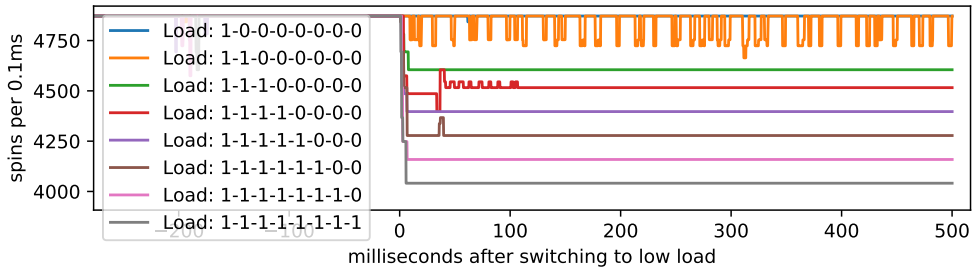


Figure 3.4: Core frequency under increasing load with different load configurations. Load is increased from zero load cores to the used load configuration. It can be observed that the frequency drops to a resting level within 5.7 ms to 7.9 ms.

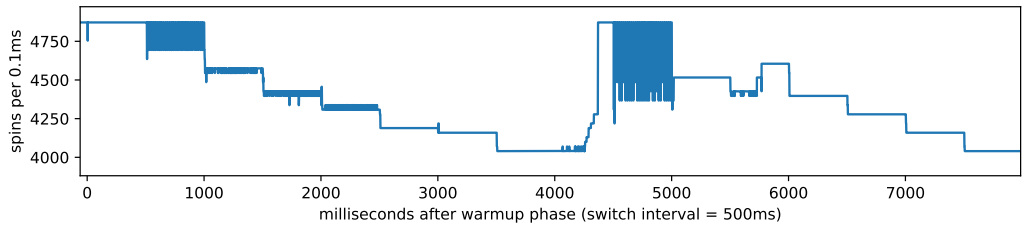


Figure 3.5: Core frequency under increasing load. The load switches from one load configuration to the next higher-load configuration every 500 ms. The set of load configurations is the same as in Figure 3.4. Switching between similar load configurations seems to decrease the chance of the measurement core reaching its resting frequency compared to when jumping between a load configuration and no load. This can be seen in the second half, where the third load configuration does not induce a stable resting frequency.

3.2.3 Test 3: Falling load

The following few tests measure the frequency change when decreasing load over time. Figure 3.6 shows the frequency change when load is decreased from a specific load configuration to zero load cores. As the graph shows, some reactions can be observed after 7.8 ms but the resting levels are reached only after 257 ms to 368 ms. This latency provides a challenge in the construction of the covert channel, as simply using the most extreme levels, i.e. 4870 and 4040 spins per 0.1 ms, as symbols requires the sender to wait up to 368 ms after disabling load before sending a new symbol.

Figure 3.7 shows that decreasing load slowly by iterating through different load configurations does not prevent the measurement core from reaching a resting frequency for each load configuration. The amount of times that the measure-

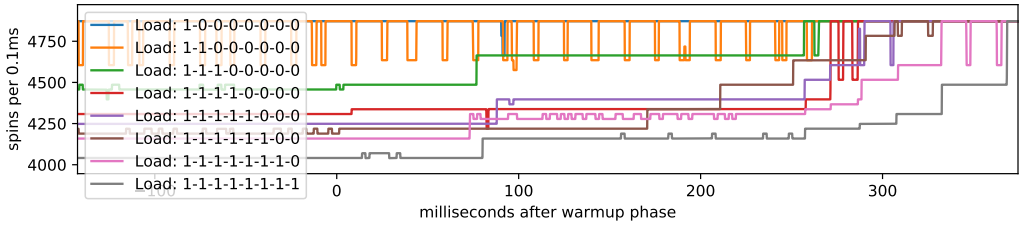


Figure 3.6: Core frequency under decreasing load with different load configurations. Load starts out at the used load configuration and is reduced to zero load cores at the 0 ms mark. The rising edge shows a far greater latency than the falling edge.

ment core oscillates between two frequency levels is increased compared to when switching between a single load configuration and no load. This increases the range of frequencies associated with a single load configuration and makes load configurations that are very similar unsuitable as symbols in a covert channel as any attempt to distinguish between them is very susceptible to noise.

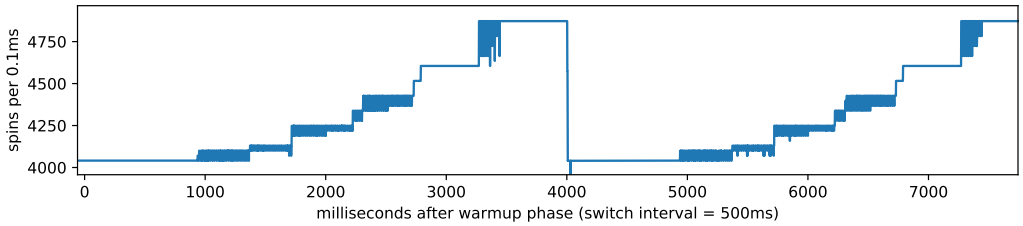


Figure 3.7: Core frequency under decreasing load. The load switches from one load configuration to the next higher-load configuration every 500 ms. The set of load configurations is the same as in Figure 3.4. The resting frequencies are always reached but many load configurations cause oscillations between two frequency levels.

3.2.4 Test 4: Alternating load

The following tests measure the frequency when repeatedly switching between low-load and high-load phases. As Figure 3.8 and Figure 3.9 show, switching between low-load and high-load phases too quickly causes the measurement core to not reach its maximum turbo frequency.

Figure 3.9 also features a slight increase in frequency with a lower latency for a few of the configurations:

- After 3 ms for the configurations `1-1-1-0-0-0-0-0`, `1-1-1-1-0-0-0-0`, and `1-1-1-1-1-0-0-0`.
- After 137 ms to 153 ms for the configurations `1-1-1-1-0-0-0-0`, `1-1-1-1-1-1-0-0`, `1-1-1-1-1-1-1-0`, and `1-1-1-1-1-1-1-1`.

These timing and amplitude of these slight increases in frequency is, however, not reliable, as repeated tests show. They are therefore not suited for use in a covert channel.

Figure 3.10 shows the frequency change when slowly increasing and then decreasing the load with a manually chosen set of load configurations. These load configurations are `1-0-0-0-0-0-0-0`, `1-1-1-1-0-0-0-0`, `1-1-1-1-1-0-0-0`, and `1-1-1-1-1-1-1-1`. They are chosen because they produce roughly equidistant resting frequencies, as shown in Figure 3.2, making this set of load configurations the best candidate for a set of symbols with size 4. The necessary time spent waiting required to reach the resting frequencies of these load configurations reliably is more than twice as high compared to when using only the most extreme load configurations, i.e. highest and lowest load. The covert channel will thus either have to use a lower symbol rate or resort to a set of two load configurations.

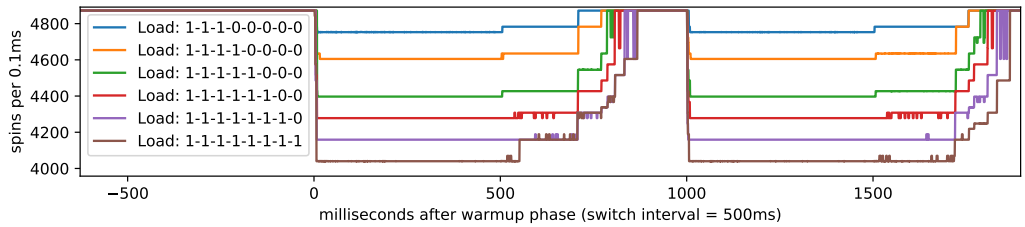


Figure 3.8: Core frequency under alternating load with different load configurations. The load is changed every 500 ms. The measurement core returns to the maximum turbo frequency in low-load phases.

3.2.5 Power usage

As seen in Figure 3.11a, the package power usage drops immediately when disabling load. Since both the amount of active cores and the package power usage respond very quickly to the load processes becoming idle, we conclude that Precision Boost 2 uses at least one other metric with high latency to determine how far the frequency multiplier can be raised. We speculate that this metric might be the current CPU temperature or a short-term TDP budget that needs to be replenished before the frequency multiplier is raised.

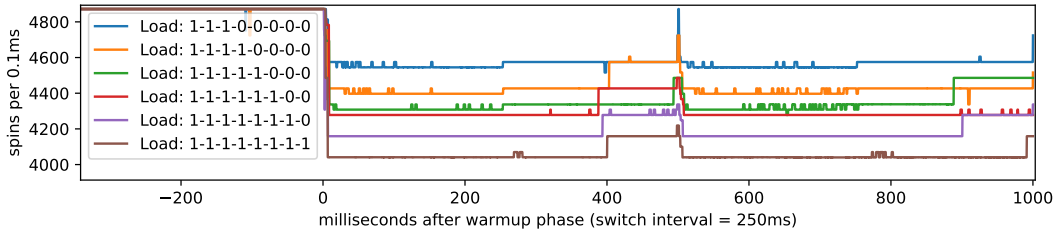


Figure 3.9: Core frequency under alternating load with different load configurations. The load is changed every 250 ms. The low-load phase is too short for the measurement core to return to its maximum turbo frequency.

The core power usage of the measurement core shown in the Figure 3.11b is roughly proportional to the frequency of the measurement core when under load. It is worth pointing out that the power usage of the measurement core rises when load cores go idle. The amplitude of this increase seems to scale with the amount of load cores going idle. Because RAPL readings require a kernel module to be loaded that is deactived by default and because the power usage might not be influenced by Precision Boost 2 in a measurable way, further investigation into this phenomenon and power usage in general is deemed out-of-scope for this thesis.

3.2.6 Different load instructions

Figure 3.12 shows the difference between using `NOP` and AVX256 instructions to create load. AVX256 instructions produce more equidistant frequencies and are thus more suitable when trying to use more than two frequencies as symbols. The latency of frequency changes when switching between high-load and low-load phases does not change significantly. The type of load instruction is thus considered to be irrelevant when construction a covert channel with two different possible symbols.

3.2.7 Side Channel

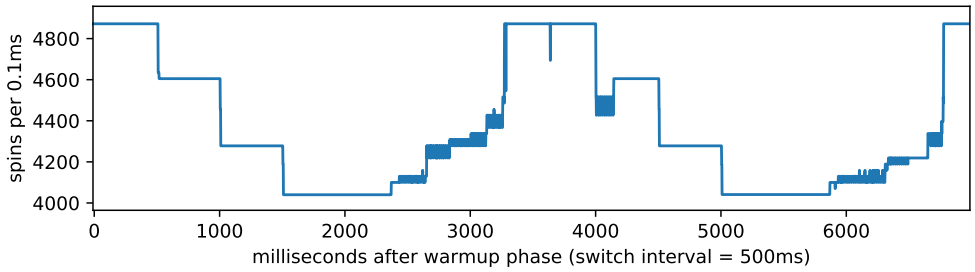
In this section, we examine whether measuring the frequency changes caused by Precision Boost 2 can be used to gain information about the activity of third-party applications. In some scenarios, this could produce a greater security vulnerability than the covert channel, for example when the activity data has sufficient resolution to count the keystrokes sent via an SSH connection, as this could allow an attacker to learn the lengths of passwords.

To determine whether the technique used in the covert channel can also be used to construct a side channel, we simulate load created by a third-party-application.

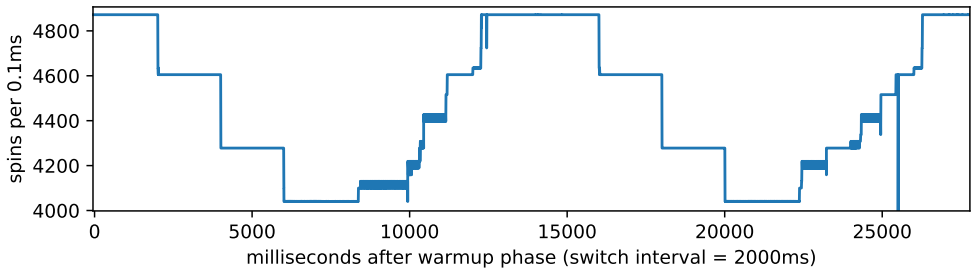
This is done by applying a load configuration and then activating an additional load process for a short amount of time. The load configuration used before activating the additonal load core will be called *base load configuration* in the context of this section. Since Precision Boost 2 only seems to lower the frequency when at least three cores are active, all of the following tests in this section use a base load configuration with at least two active cores.

As our previous tests have shown, Precision Boost 2 reacts quickly to rising load and slowly to falling load. Figure 3.13a shows that Precision Boost 2 reacts slower when more cores are active. On bursts of 100 ms, the base load configuration `1-1-0-0-0-0-0-0` shows a falling-load latency of 223 ms, which is the lowest of the tested base load configurations. This base load configuration is thus optimal when trying to observe the activity of third-party applications. As shown in Figure 3.13b, longer activity bursts cause longer rising-edge latencies. This difference in latency becomes less extreme with higher load burst times.

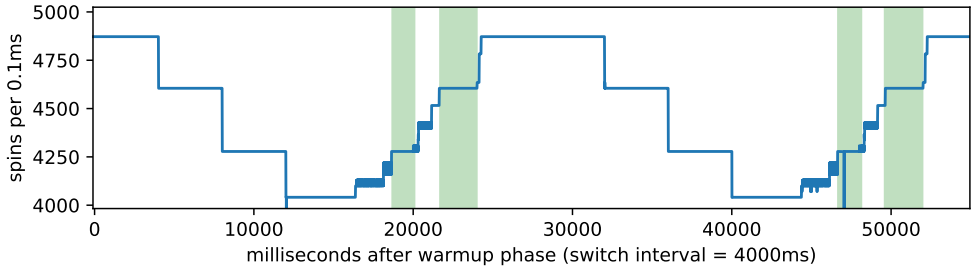
The technique presented in this thesis can thus be used to observe the activity of third-party applications. When processing the frequency measurements automatically, the length of a load burst has to be considered when compensating for latency. Because of the high latency, multiple load bursts in quick succession might falsely be identified as a single, longer load burst.



(a) Each load configuration is applied for 500 ms.

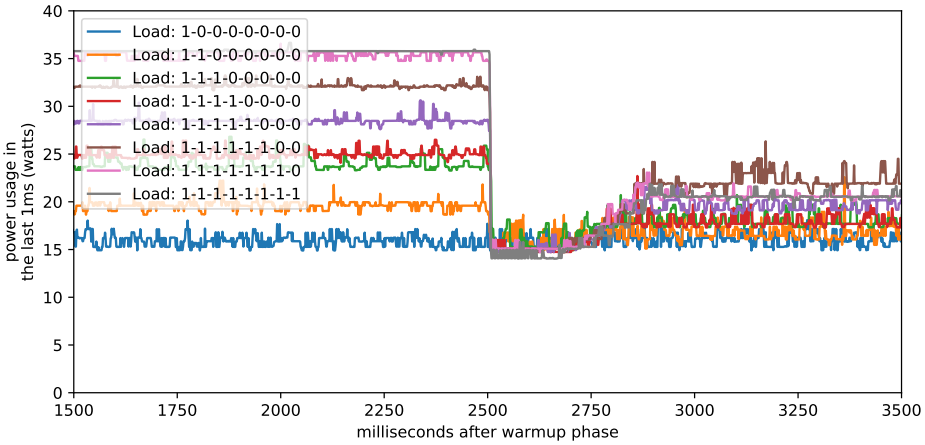


(b) Each load configuration is applied for 2000 ms.

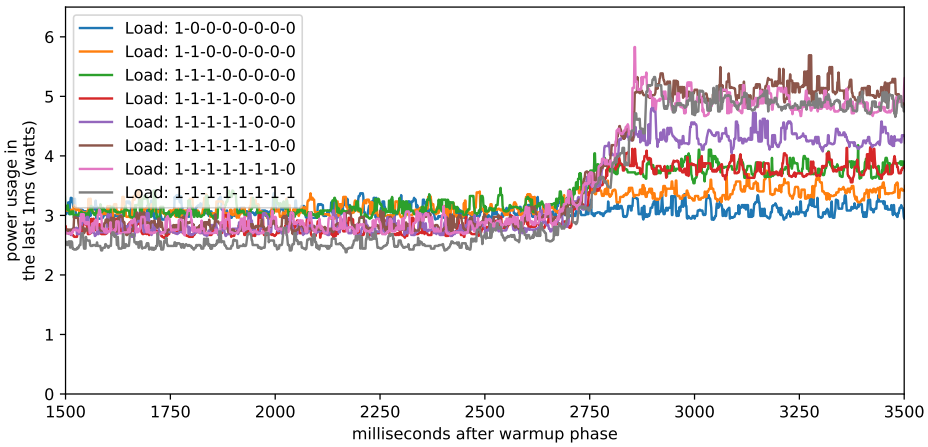


(c) Each load configuration is applied for 4000 ms.

Figure 3.10: Core frequency when increasing and then decreasing the load twice via several intermediate load configurations. When decreasing load every 500 ms or 2000 ms, the resting frequencies are not reached reliably. Increasing the length that each load configuration is applied to 4000 ms solves this issue, as shown in (c). The phases in which the resting frequency is observed while decreasing load are highlighted.

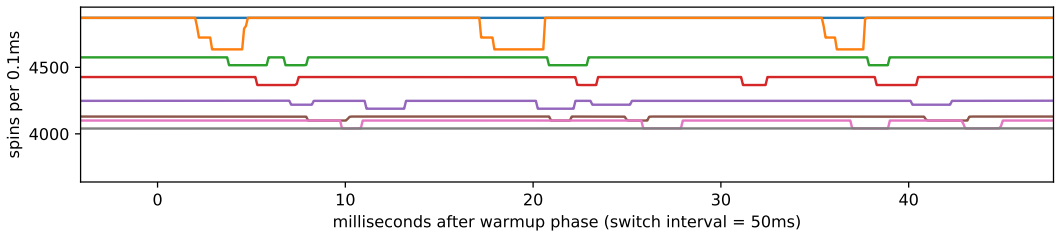


(a) Package power usage

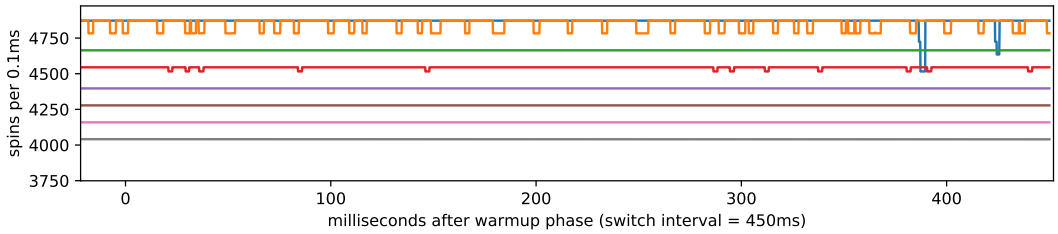


(b) Core power usage

Figure 3.11: Core and package power usage with different load configurations. The load configuration is active until the 2500 ms mark. After that, all load cores are idle. Note that the power usage axis is scaled differently in the two plots.

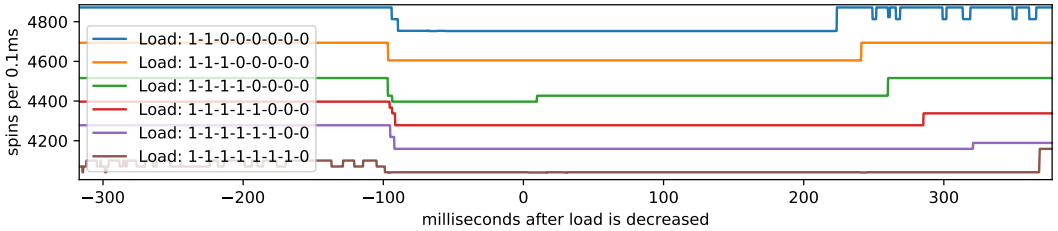


(a) Using `NOP` instructions.

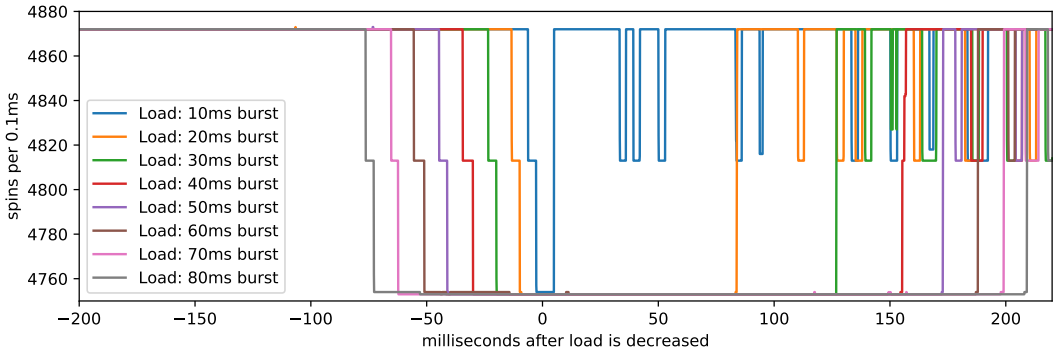


(b) Using `VADDPS` instructions.

Figure 3.12: Core frequency under constant load with different load configurations. To create load, a sequence of `NOP` instructions are used in (a) and a sequence of AVX256 instructions are used in (b).



(a) Rising-edge latency increases with the amount of active cores. The load configuration `1-1-1-1-0-0-0-0` shows a faster first reaction.



(b) Longer load burst times cause higher rising-edge latencies.

Figure 3.13: Core frequency when activating one additional load process for a short burst and then falling back to the previous load configuration. The additional load process is deactivated at 0 ms.

Chapter 4

Design & Implementation

The covert channel designed in this thesis allows communication between two processes, called sender and receiver, that are not allowed to communicate by system security policies and can thus not rely on operating system support. Utilizing the dynamic frequency scaling technology AMD Precision Boost 2, the sender manipulates the frequency of the core that the receiver runs on by changing the load applied to the other cores. The design approach presented in this thesis is very similar to that of the covert channel based on Intel Turbo Boost presented by Kalmbach [8], with changes addressing the unique problems faced when dealing with Precision Boost 2.

4.1 Approach

As observed during the analysis in Chapter 3, applying load to more than two processor cores makes Precision Boost 2 lower the turbo frequency multiplier, reducing the core frequency of the active cores. By branching into multiple processes or threads, the sender process can control the amount of cores that are under load. The receiver can then measure the frequency by repeatedly increasing a *spin counter* in a loop and measuring how far the counter can be increased within a set time frame. This spin counter is then used as a measure of core frequency.

Amount of symbols. As the turbo frequency multiplier tweaked by Precision Boost 2 can take many different values, we theoretically have more than two different frequencies that the sender can induce on the receiver's core. We thus have to distinguish between *symbol rate* and *bit rate*. The symbol rate measures the frequency with which the sender changes the applied load and thus the core frequency of the receiver's core. Every load configuration that induces a distinct frequency on the receiver's core is a symbol supported by the covert channel. The bit rate measures the amount of information transmitted between the sender and

receiver in a set time frame. In order to map symbols to bits and thus link symbol rate and bit rate, we have to choose the set of possible symbols. A larger set of possible symbols means that each symbol carries more information. Assuming that the size of the set of possible symbols n is a power of two, each symbol carries exactly $log_2(n)$ bits of information.

Unfortunately, as discovered in Section 3.2.4, jumping between similar frequencies without returning to a no-load configuration causes the dynamic frequency scaling to become less reliable. This can be mitigated by increasing the time between load configuration changes. Since this is effectively a reduction of the symbol rate, we have to balance symbol rate and amount of possible symbols. Our tests during the analysis, specifically Figure 3.10, show that the symbol rate has to be more than halved to be able to support four distinct frequencies. We thus choose to only use the two most extreme, i.e. highest and lowest, frequencies as this makes distinguishing them easier. This has the additional advantage of increased simplicity, as each symbol carries exactly one bit of information and our symbol rate equals our bit rate. In this thesis, we assign a low load and thus high frequency to a logical `1` and a high load and thus low frequency to a logical `0`.

Asynchronous communication. Since neither the sender nor the receiver can control the operating system's scheduler and because there is no secondary communication channel, this covert channel will, similar to the covert channel presented by Kalmbach [8], not rely on the sender and the receiver being synchronized. Instead, the sender will prepend its messages with a special value, called the *magic value*, as shown in Figure 4.1. The sender can then identify the start of a message by simply listening on the channel until this specific sequence of bits is read. The magic value could theoretically be anything but to minimize false-positives, the value should be chosen such that the chance of encountering the bit sequence as part of a message body or part of the noise is minimized. Since

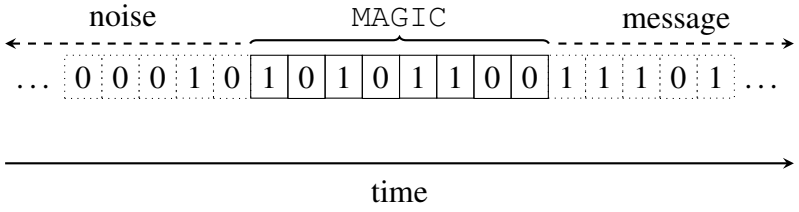


Figure 4.1: The magic byte is prepended to every message, allowing the receiver to detect the start of a message without needing to synchronized with the sender beforehand.

large sequences of zeroes of ones are quite common when sending binary data, the magic value should not consist entirely of ones or zeroes or longer sequences thereof. A longer magic value decreases the chance of false-positives but also

increases the overhead per message. In this covert channel, the magic value will be exactly one byte in size, as this makes implementation easy. As in Kalmbach's work, the bit sequence is chosen to be `10101100` because it contains a large number of changes between one and zero and also contains a sequence of ones and zeroes, which is assumed to rarely be included in the noise or message body.

4.2 Error detection & correction

Scheduler decisions heavily influence the performance of the covert channel. To be able to communicate at all, both sender and receiver must be active at the same time. Every time the scheduler pauses the receiver process, the spin counter drops, possible even below the threshold value used to distinguish the two frequencies. This introduces bit errors, i.e. bits in the received message that are flipped compared to the sent message. The scheduler can also interfere with the sender but since the sender is split into multiple processes, it is unlikely that all sender processes are paused at the same time. A reduced amount of active sender processes will cause a lower load and thus a higher frequency in the receiver's core. By setting the threshold value high enough, the covert channel can be made resistant to this type of scheduler-induced noise.

Transfer protocol. Even after tweaking the threshold value, bit errors may still occur. To ensure that messages can still be transmitted reliably, we have to include the means for the receiver to either correct errors, such as error correction codes, or to detect errors and request retransmission, as is possible when using TCP. In order to evaluate the performance of the covert channel in comparison to Kalmbach's covert channel based on Intel Turbo Boost [8], this thesis will also use a simplified version of TCP to deal with transmission errors. To achieve this, messages are split into packets, as shown in Figure 4.2. Each of those packets contains the magic value, a packet sequence number used to identify the packet within a single stream of packets, the actual message data and a CRC16 checksum. This checksum will take the bit sequence including the packet sequence number and the data as input. Since the magic value will always the same on a detected packet, including it in the checksum input would be detrimental to the checksum check accuracy. To avoid having to include length information in the packet, all packets will have equal size. This requires the sender to pad messages to the appropriate length, should they be too short.

Upon receiving a full packet, the receiver calculates the CRC16 checksum of the sequence number and the data field and compares it to the checksum included in the packet. Should these two values differ, the packet is assumed to be corrupt. Note that any change in the sequence number, data field or included checksum will cause this check to fail. An error in the magic value will cause the packet to not be

Figure 4.2: Simplified TCP packet structure. Each packet contains the 8-bit magic value, its 8-bit sequence number, the actual data and a 16-bit CRC16 checksum.

recognized at all, which is out-of-scope for the checksum check. Theoretically, a packet with multiple bit errors could contain both a corrupt sequence or data field and a corrupt checksum field that happen to match up, resulting in an undetected error but the chance of that happening is deemed negligible in the context of this thesis.

Whenever a packet is received successfully, the receiver sends an acknowledgement packet (ACK) back to the sender. This is effectively a reversal of roles, so the same code used for the sender can be used in the receiver and vice versa. Upon receiving the ACK for a packet with sequence number n, the sender starts sending the packet with sequence number $n + 1$, assuming that the receiver has already received all preceding packets successfully. As the only type of packet the receiver will send are ACKs, the packets don't have to contain any kind of control bits signalling that they are ACK packets. The ACK packets can thus consist entirely of the magic value and the packet sequence number of the last successfully received packet. Note that there is no checksum in the ACK packet, as this would double the packet size.

The lack of a checksum opens up the possibility that the sender will receive a corrupt acknowledgement. A corrupt acknowledgement contains a packet sequence number that is either too high or too low. If the sequence number is too low, then the sender re-transmits a data packet that the receiver already received successfully. In response to this, the receiver simply discards that data packet and retries the acknowledgement of the correct data packet. If the sequence is number too high, then the sender transmits a data packet further down the packet sequence, leaving the receiver with a gap in the list of successfully received packets. Again, this is fixed by the receiver retrying the acknowledgement. The sequence number might even be so high that it acknowledges a data packet that the sender has not yet sent. In this case, the corruption is detected by the sender and the sender simply re-sends the last data packet that was sent.

Note that it is possible that the ACK packet's magic value gets corrupted or that the sender gets paused by the scheduler during transmission, causing it to miss the ACK packet entirely. To avoid a deadlock, the sender starts a timeout for the ACK packet after it finishes sending the data packet. If the timeout is triggered before an ACK packet is read, the sender re-transmit the last data packet.

When choosing how long the ACK timeout should be, the bit rate B (in bits per second), ACK packet size A (in bits) and channel latency L (in milliseconds) have to be considered. Assuming the receiver starts sending the ACK packet right after receiving the data packet, the receiver will start sending with a delay of L and then spend the time $\frac{A}{B}$ sending. The sender will receive the ACK packet with a delay of $\frac{A}{B} + 2L$. To include slight tolerances for varying latency, the ACK timeout is set to $\frac{A}{B} + 4L$.

Dynamically increasing ACK timeouts. Note that with this ACK timeout, deadlocks can still occur in the following scenario:

1. The sender sends a data packet.
2. The receiver misses the data packet's magic value.
3. The packet data field contains the magic value as a bit sequence and the receiver mistakes it for the start of a packet.
4. The sender finishes sending the data packet and starts waiting for the ACK packet.
5. The receiver is still receiving this *phantom packet* when the ACK timeout is triggered.
6. The sender sends the data packet again.
7. The receiver finishes reading the phantom packet, concludes that it is a corrupt packet and sends an ACK for the last successfully received data packet.
8. The receiver finishes sending the ACK packet. Because ACK packets are very small, this can occur before the sender has sent the part of the data field that contains the magic value.
9. The receiver mistakes the bit sequence included in the packet data field as the magic value again, repeating the cycle.

In order to break free from this cycle, the sender doubles its ACK timeout whenever the timeout it triggered. Upon successfully receiving an ACK packet, the timeout is reset to the initial value. While the symbol rate, frequency threshold, and packet size can theoretically be determined dynamically by having the sender and receiver run tests, this thesis contains simple implementations with hard-coded values.

4.3 Sender implementation

The goal of the sender is to send a message by applying high load to send a logical `0` and low load to send a logical `1`. This is achieved by spawning seven additional load processes that can be controlled from a seperate control process. While it is possible that the scheduler moves the sender's control process onto an otherwise idle processor and thus keeps one more core active than intended, the analysis in Chapter 3 has shown that Precision Boost 2 does not reduce the turbo frequency multiplier before three cores become active, so this is unlikely to have a detrimental effect on the covert channel. The load processes communicate with the control process via shared memory regions and semaphores used to wake up sleeping load processes. While this might seem to contradict our goal of not relying on operating system support, disabling forking, shared memory or semaphores is not seen as a realistic countermeasure, making this an acceptable dependency on operating system support.

After creating the load processes, the message is split into the necessary amount of TCP packets, which are then sent by applying load. When ordered to apply load, the load processes run a sequence of AVX256 instructions, specifically the `VADDPS` instruction, in a loop. Since the loop also contains condition check and jump instructions, the actual instruction sequence does not consist entirely of `VADDPS` instructions. To mitigate any effects that this might have, the length of each loop iteration is increased by executing 300 `VADDPS` instructions before reaching the end of the loop. This number is chosen arbitrarily but is deemed sufficient to make the effects of the condition check and jump instructions negligible. When starting to send a logical `1`, all load cores are sent to sleep using the `sem_wait` C function on their assigned semaphores. They are later woken up when the control thread calls `sem_post` on their semaphores. The sender control process itself is inactive while load is applied. This is achieved by sending the control process to sleep for the duration of a single symbol, as shown in Figure 4.3.

The sender keeps track of the packets that have already been sent, so that corrupt ACK packets acknowledging data packets that have not yet been sent can be detected.

4.4 Receiver implementation

The goal of the receiver is to measure the current core frequency and reconstruct the bit stream sent by the sender. The core frequency is measured with a spin counter and a hard-coded threshold value is used to distinguish between low and high frequencies.

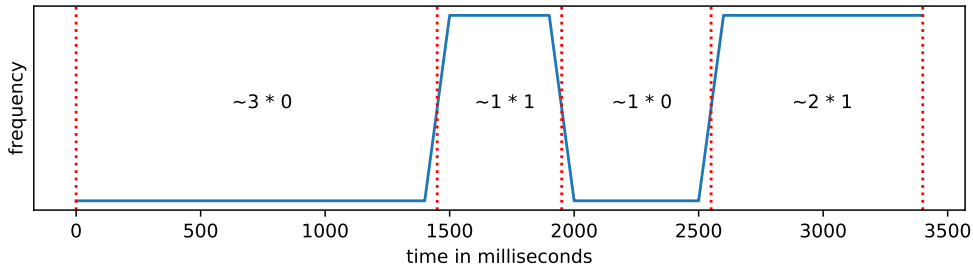
```
1 void send_bit(const int bit) {
2     int *cur_mask = bit ? one_mask : zero_mask;
3
4     set_load(cur_mask);
5     usleep(BIT_LENGTH_MS * 1000);
6 }
```

Figure 4.3: Function used to send a single bit. After applying the appropriate load configuration, the sender control process goes to sleep for the duration of one symbol.

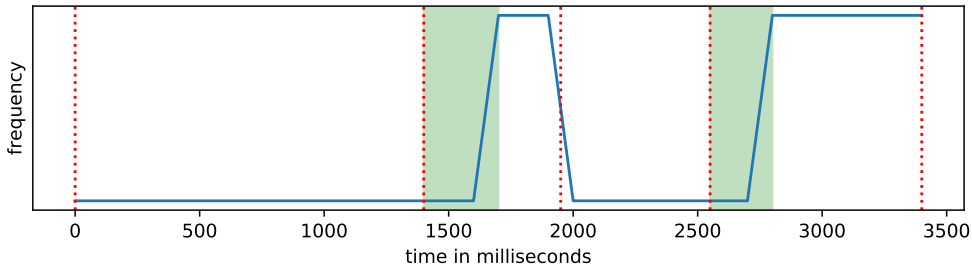
Since the covert channel does not feature a clock signal, translating the frequency samples into a bit stream is not trivial. Simply measuring the frequency with a set sampling rate adjusted to the symbol rate is susceptible to slight delays introduced by the scheduler pausing either the receiver or sender that can add up over time. A simpler solution presented by Kalmbach in his covert channel based on Intel Turbo Boost [8] uses edge detection to identify symbol changes and the length between those changes. Because of the very asymmetric latencies produced by Precision Boost 2 when switching between high load and low load, this thesis introduces a slight change to the edge detection method.

In general, the edge detection technique works by storing the time of the last symbol change and, upon detecting another symbol change, calculating the amount of time that the previous symbol has been transmitted. Since the symbol rate is fixed, the amount of time spent on sending a single symbol, called the symbol length, is known. The length observed by the edge detection is then rounded to the closest multiple of the symbol length, as shown in Figure 4.4a. When working with Precision Boost 2, the rising edge has a very high latency compared to the falling edge, causing the high-frequency phases observed by the receiver to be shorter than intended by the sender. This can be accounted for by adding the latency to the length of high-frequency phases and subtracting the latency from the length of low-frequency phases, as shown in Figure 4.4b. This *latency correction* is performed by the receiver.

Since edge detection only reads bits whenever an edge is encountered, the receiver might get stuck if the sender ends its message on a high-frequency and no other third-party applications cause a high load. To prevent this, the receiver will insert an artifical *double-edge* whenever it reads a large sequence of high-frequency symbols. The receiver does this by inserting a low-frequency phase of minimal width into the stream of measured frequencies. This double-edge produces a very small low-frequency phase that gets rounded down to zero symbol lengths, which does not affect the read bit stream. This technique is illustrated in



(a) edge detection with symmetrical latencies

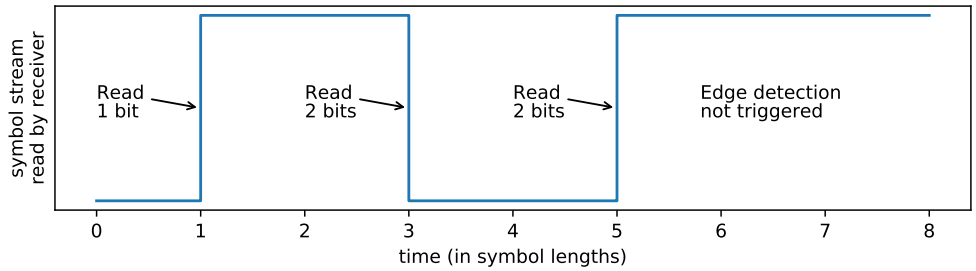


(b) edge detection with asymmetric latencies and latency correction

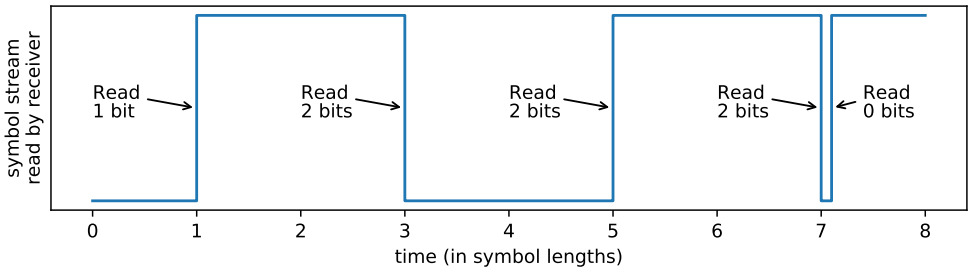
Figure 4.4: Sample frequency reading made by the receiver. Edge detection is used to reconstruct the bit stream. (b) shows how asymmetric latencies can be accounted for by increasing the length of the previous symbol sequence whenever the high-latency edge is encountered.

Figure 4.5. This problem could also be solved by having the sender send an additional bit after it finished transmitting the message. This could, however, cause the sender to miss the start of an ACK packet if the receiver didn't require the additional bit and started sending the ACK packet immediately. We thus choose to insert double-edges instead.

The receiver measures the frequency by increasing a spin counter for 0.1 ms. In order to do this, the receiver needs a way of accurately measuring time. A standard way of achieving this is by repeateadly calling the `gettimeofday` C function. This, however, requires operating system support and makes the covert channel susceptible to relatively non-intrusive countermeasures such as adding noise to timings returned by operating system functions or detecting processes using the covert channel because they are querying the time very frequently. A better approach is reading the CPU-internal *Time Stamp Counter* (TSC) with the `RDTSC` instruction. This counter is increased with a constant frequency, independent of frequency scaling technologies like Precision Boost 2. The frequency with which the TSC is increased can be measured by calling `gettimeofday`



(a) bitstream ending on high frequency



(b) bitstream with artificial double-edge

Figure 4.5: Bitstream received by sender that ends on a high frequency. In (a), the last bits are not read because edge detection is not triggered. Inserting an artifical double-edge into the read symbol stream fixes this problem, as shown in (b).

twice and measuring the TSC difference between those timings. Since the TSC frequency, which was above 3000 MHz in our tests, is significantly higher than our spin counter resolution (10 kHz), slight measurement errors when determining the TSC frequency do not affect the performance of our covert channel. Using the TSC does present a few challenges, though.

The TSC is not synchronized across cores which is a problem when the scheduler moves the receiver control process to a different core, as this causes the timing calculations to return incorrect values. Assuming that the receiver is not moved across cores multiple times per millisecond, this interference will only affect a single spin counter measurement and thus produce a frequency drop or spike for only 0.1 ms, which gets filtered by the edge detection.

The other problem when using the `RDTSC` instruction is that it is not a serializing instruction and as such is subject to instruction reordering by the processor. To avoid any problems that might be caused by this property, the `RDTSCP` instruction, a serializing variant of `RDTSC`, is used instead.

Chapter 5

Evaluation

The goal of the evaluation is to determine the maximum bandwidth of the covert channel under both idealized and realistic circumstances and determine concrete parameters for the symbol rate, frequency threshold and packet length with which this bandwidth is reached. These metrics are explained in Section 5.1, followed by a description of the setup used throughout the evaluation in Section 5.2. In Section 5.3, we determine the optimal symbol rate, frequency threshold and packet length by measuring and minimizing the error rate. Afterwards, we measure average gross and net bit rate of the covert channel in Section 5.4. As our covert channel might be used on systems on which other third-party applications produce enough load to affect the frequency scaling, we examine the resistance of the covert channel to these kind of disruptions by other processes in Section 5.5. Finally, countermeasures that can be implemented to close or slow our covert channel are discussed in Section 5.6.

5.1 Metrics & Parameters

In this section, we describe the metrics and parameters used in the tests made throughout the evaluation. The parameters, such as the symbol rate, are first optimized with regards to the error ratio that they produce and are later used as parameters when determining the channel bandwidth.

Gross bit rate. The total amount of bits transferred through the covert channel per second, including overhead introduced by the transfer protocol, such as the magic value, the packet sequence number, the checksum and ACK packets.

Net bit rate. The amount of data bits, i.e. bits of the message that is to be sent, transmitted per second. This does not include overhead introduced by the transfer protocol.

Symbol rate. The amount of symbols transmitted per second. Since our covert channel supports only two different symbols, each symbol carries exactly one bit of information, causing our symbol rate to equal the gross bit rate.

Symbol length. The amount of time between symbol changes. This is derived from the symbol rate. Longer symbol lengths are expected to produce lower error ratios as there is more time for the frequency to stabilize. Shorter symbol lengths increase the symbol rate and thus the gross bit rate.

Frequency threshold. A concrete frequency value used to distinguish low frequencies from high frequencies. In our covert channel, the minimum and maximum frequency that can be induced on the receiver's core by applying load are used as symbols. Third-party applications can disrupt the covert channel by increasing the load and thus lowering the frequency on the receiver's core. A frequency threshold closer to the minimum frequency is thus excepted to increase the covert channel's resistance against these kind of disruptions.

Packet length. The length of a packet used in the transfer protocol of the covert channel. As the magic value, sequence number, and checksum fields have a fixed size, the packet length can only be varied by increasing or decreasing the amount of data bits sent with each packet. Longer packet lengths decrease the protocol overhead per message. Shorter packet lengths decrease the chance that a packet gets corrupted during transfer, reducing the amount of re-transmissions.

5.2 Setup

The installed hardware and software is the same as in previous sections, featuring an AMD Ryzen 7 2700 and the Arch Linux distribution running the Linux kernel on version 5.1.16. Like in the analysis, we focus on keeping a minimal set of installed software packages and choose the `performance` frequency governor over `schedutil`. In contrast to the setup in the analysis, the `isolcpus` kernel parameter is not used and the sender and receiver make no use of the `sched_setaffinity` function. Instead, the processes are created without setting the affinity masks, letting the scheduler determine the core assignments. During our tests with idealized circumstances, no additional background services expect for a Secure Shell (SSH) daemon are active. We simulate workload produced by third-party applications during our tests on noise resistance in Section 5.5 by creating processes that produce load in random intervals. During our tests, the sender and receiver control process measure the metrics mentioned above and save them together with parameters such as symbol length, the message that is to

be sent, and the frequency threshold and write them to seperate files that are then analyzed by us.

5.3 Parameters & Error ratio

In this section, we determine the optimal frequency threshold, symbol length and packet size. The frequency threshold is chosen by repeating some of the measurements made during the analysis in Chapter 3. Since we removed stabilizing settings such as the `isolcpus` kernel parameter, simply re-using the results from the analysis is susceptible to changes introduced by letting the scheduler determine the core assignments. The symbol length and packet size are chosen based on the error ratios that they produce, similar to the approach taken by Kalmbach during the evaluation of this covert channel based on Intel Turbo Boost [8].

The errors that we encounter during transmission are similar to those encountered by Kalmbach. When trying to understand these errors, we distinguish between *bit errors* and *synchronization errors*. Bit errors occur whenever a bit gets flipped during transmission. In our case, this can happen when the frequency threshold is not set correctly, causing the receiver to incorrectly identify the load applied by the sender. Bit errors can also happen when third-party applications produce additional load during phases when the sender intends to induce a high frequency on the receiver's core. Synchronization errors occur whenever a bit gets lost in transmission or a single bit is incorrectly read as two bits. This can happen when the scheduler pauses the receiver, causing it to notice a frequency switch with a high delay. During reconstruction of the bit stream with edge detection, we round the time that the last frequency level was active to the nearest multiple of the signal length. High delays in noticing frequency changes can cause the receiver to miscalculate the amount of bits sent since the last frequency change. The same effect can occur when the latency of frequency changes varies too strongly from the expected value, as this causes the latency correction to under- or overcompensate.

5.3.1 Frequency threshold & Latency correction

Re-running the measurement shown in Figure 3.2 with the two most extreme load configurations, i.e. highest and lowest load, and applying the median filter returns the results shown in Figure 5.1. Following our speculation that a low frequency threshold increases resistance to third-party applications applying additional load, the frequency threshold could be set to 4200, close to the lower frequency.

Unfortunately, setting the frequency threshold this low produces synchronization errors in every message. As shown in Figure 5.2, the frequency rises across several intermediate levels before reaching a resting level. A low frequency thresh-

old thus causes the receiver to notice the frequency switch earlier. Since the latency correction is set up with the expectation that the edge is detected once the resting frequency is reached, this low frequency threshold causes the latency correction to overcompensate, adding more `0`-bits to the read bitstream than intended. Adjusting the static latency correction to a lower frequency threshold does not produce reliable results because the latency with which the intermediate frequency levels are reached varies even when using the same load configuration, as shown in Figure 5.2. The frequency threshold must thus be chosen so that it is reached just before the frequency rises to the resting level. In our measurements, the last intermediate level reliably lies between between 4480 and 4750 spins and the resting frequency is on 4870 spins. We thus place the frequency threshold in the middle of the resting frequency and the maximum last intermediate frequency, at 4810 spins.

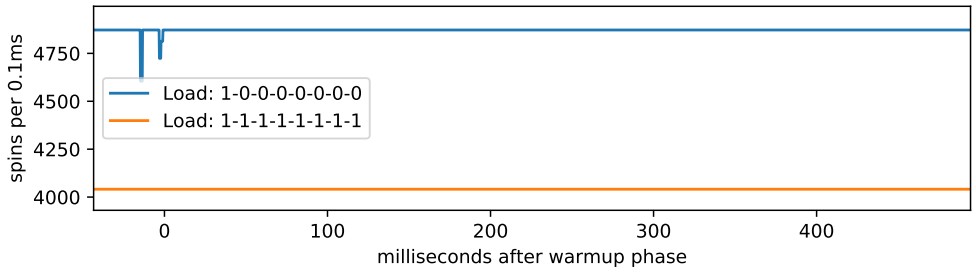


Figure 5.1: Frequency of the receiver's core under minimum and maximum load. The minimum frequency rests at about 4040 spins while the maximum frequency rests at about 4870 spins.

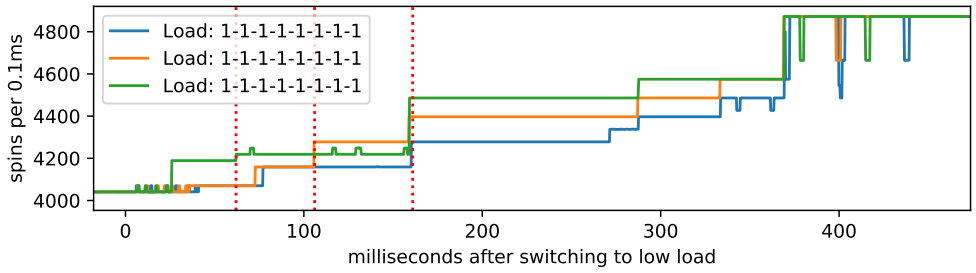


Figure 5.2: Multiple frequency measurings when changing from maximum load to mininmum load. The dotted lines mark the mark the times at which one of the measured frequencies surpasses 4200 spins.

5.3.2 Symbol length

Our analysis in Chapter 3 has shown that the latency of the rising edge when using the load configuration `1-1-1-1-1-1-1-1` is about 368 ms. We test symbol length of 360 ms to 450 ms in 10 ms steps and measure the error ratio when transmitting 40 randomly generated 8-bit messages. We define the error ratio as the ratio of the amount of incorrectly transmitted messages vs. the total amount of transmitted messages. Figure 5.3 shows the result of this test. Lower symbol lengths are preferable because they increase the symbol rate and thus the gross bit rate of our covert channel. We thus choose 400 ms, the smallest symbol length with a 0 % error ratio.

Note that during Kalmbach's tests [8], increased symbol lengths caused the amount of synchronization errors to rise. He argues that higher symbol lengths allow more disruptions to occur during the transmission of each symbol. In our tests, however, higher symbol lengths decrease the error ratio. We speculate that this is due to the fact that our possible symbol lengths are an order of magnitude higher than those used by Kalmbach. This, combined with the fact that Precision Boost 2 reacts slower than Intel Turbo Boost, make long symbol lengths less error-prone, as short-term disruptions can be filtered out.

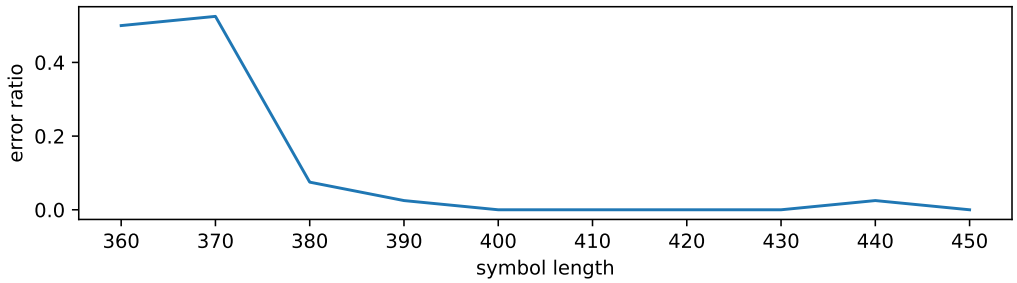


Figure 5.3: Error ratio when sending an 8-bit message with different symbol lengths, repeated 40 times. The symbol lengths greater than or equal to 400 ms produce minimal error ratios. We deem the $\frac{1}{40}$ error ratio at 440 ms a negligible fluctuation.

5.3.3 Packet size

Even with a symbol length that produced a 0 % error ratio when sending 8-bit messages, we expect longer messages to be more susceptible to bit errors and synchronization errors. A larger packet size thus increases the chance of reading a corrupt packet and having to re-transmit. Lowering the packet size increases

the protocol overhead, reducing the net bit rate. To find an optimal value, we test multiple packet sizes in regards to the error ratio that they produce and choose the largest one with an acceptable error ratio. As Figure 5.4 shows, the packet size 12 bytes is the largest packet size that produces an error ratio of 0 %, with higher packet sizes quickly raising the error ratio.

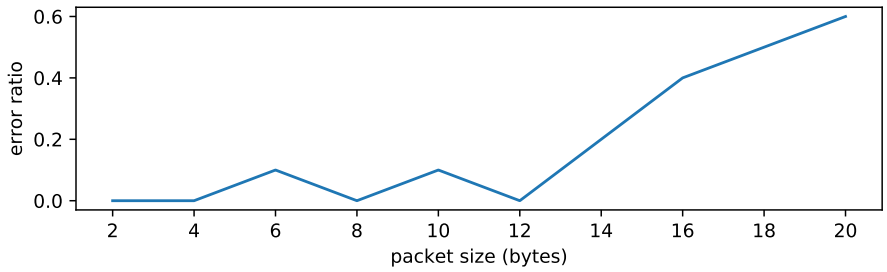


Figure 5.4: Error ratio when sending messages of different sizes.

5.4 Bit rate

Using the frequency threshold, symbol length and packet sizes determined in the sections above, we now measure the average gross bit rate and net bit rate by sending a randomly generated 96 byte message over the covert channel and measuring the time it takes until the receiver has successfully received the entire message. This message is split into packets according to our transfer protocol and corrupt packets are re-transmitted until the receiver received them successfully. The table in Figure 5.5 shows the results of this transmission.

Metric	**Value**
Length of message excl. overhead	96 bytes
Time until the message was transmitted	708 seconds
Net bit rate	1.08 bit/s
Length of message incl. packet overhead	144 bytes
Length of message incl. packet and ACK overhead	168 bytes
Gross bit rate	1.90 bit/s
Amount of re-transmissions	0

Figure 5.5: Result of transmitting a 96 byte message.

5.5 Noise resistance

To determine the resistance of the covert channels against noise produced by third-party applications, we measure the error ratio when sending 40 8-bit messages while simulating load produced from other applications in the background. This load is produced by spawning additional load processes and keeping them active for the entire duration of the transmission. These additional load processes will be called *noise processes*. As shown in Figure 5.6, our current implementation fails to transmit any message successfully as soon as noise processes are activated. This is because the frequency threshold is set too high to distinguish between two non-maximum frequencies. Note that this threshold is only set this high in order to make the latency correction reliable, as explained in Section 5.3.1. The covert channel can be made more resistant to noise by disabling latency correction and lowering the frequency threshold. In order to correctly identify symbol lengths without latency correction, the symbol length has to be increased to at least twice the rising-edge latency. This way, the rounding implemented in our edge detection filters out the rising-edge latency. Increasing the symbol rate, however, would decrease the bit rate of the covert channel. An attacker thus has to balance noise resistance and bandwidth.

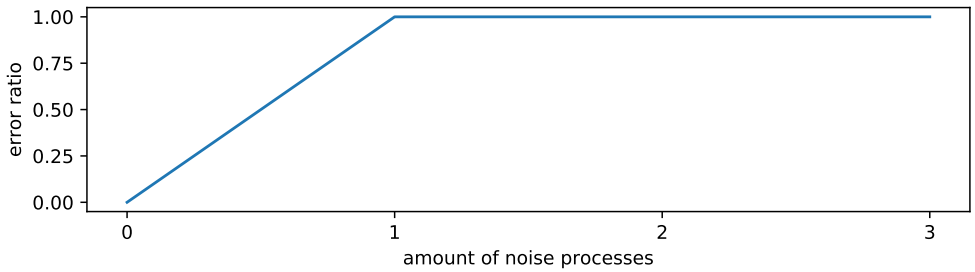


Figure 5.6: Error ratio when sending 40 8-bit messages with different amounts of active noise processes.

5.6 Countermeasures

In this section, we take a look at possible countermeasures that can be implemented to close or slow the covert channel presented in this thesis. All of the countermeasures presented in this section cause a performance penalty for legitimate applications. A hardware and software developer or system administrator must therefore consider whether the security benefit gained by implementing these countermeasures is worth the performance penalty.

The simplest hardware countermeasure is disabling Precision Boost 2. This does, however, negate any performance benefit introduced by Precision Boost 2 and might thus not be a favourable option. Instead of completely disabling it, Precision Boost 2 could be altered so that its decisions are harder to influence. How this could be achieved, especially given that Precision Boost 2 already reacts very slowly when reducing load, remains to be investigated. As with most hardware countermeasures, implementing them would also require exchanging existing hardware, which might be a too cost-intensive measure.

More promising than hardware solutions are software countermeasures. The fact that this covert channel is built to minimize reliance on operating system support poses a challenge when building software countermeasures. Kalmbach [8] discussed setting the *Time Stamp Disable* (TSD) flag as a software countermeasure. This would make `RDTSC` a privileged instruction and allow the operating system to manipulate the read TSC values. Since legitimate applications might also be using the TSC value, completely disabling `RDTSC` or manipulating the read TSC value too heavily is not considered an option. Slight manipulations of the TSC value would cause the receiver to miscalculate the time when counting spins and thus measure the frequency incorrectly. The impact of this manipulation can be lowered by decreasing the resolution of the frequency measurements. The current resolution is 0.1 ms or 10 kHz, which is far above the 2.5 Hz frequency with which the symbol is changed. The covert channel can thus trivially be modified to have high tolerance to slight time skips.

As shown in Section 5.5, the latency correction requires the covert channel to work with a high frequency threshold, making it susceptible to noise produced by the load of other applications. To target this covert channel specifically, the operating system could create processes that randomly generate noise. This would introduce bit errors and synchronization errors that slow the transmission or even make it impossible to reliably transmit data. Note, however, that the covert channel can be adjusted to have higher resistance to this countermeasure by reducing the symbol rate and frequency threshold and disabling latency correction.

5.7 Discussion

In this section, we discuss findings made in this thesis and the limitations of the constructed prototype and their effect on the evaluation. We have successfully constructed a covert channel based on AMD Precision Boost 2, finding solutions for the challenges introduced by the slow reactions of Precision Boost 2. We compensate for the asymmetric latencies by implementing latency correction after detecting edges and examine the side-effects introduced by the latency detection.

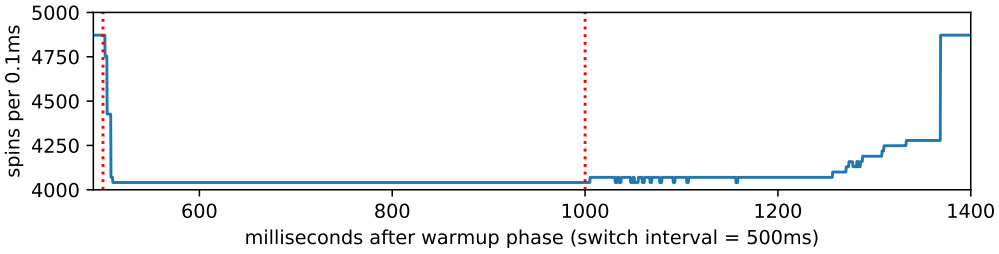
The optimal symbol rate, frequency threshold and packet size are chosen such that they produce a minimal error ratio.

More sophisticated approaches that consider a wider ranges of possible values for these parameters and judge them based on expected average throughput are possible. Dynamically determining these parameters would make the covert channel easier to use and also possibly make it able to run on systems on which third-party applications produce additional load. Possible effects that these parameters have on each other have also not been considered. The values for these parameters found during our evaluation might thus not be optimal and it is possible that the covert channel bandwidth can be increased further.

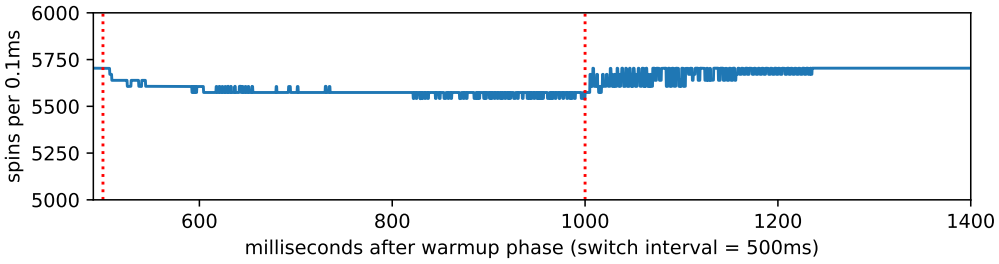
When we determined the symbol length, we did not adjust for the asymmetry of the edge latencies. The falling edge reacts within a few milliseconds, so it is possible that a higher average symbol rate can be achieved by decreasing the symbol length to a few milliseconds and waiting 368 ms whenever a rising edge is produced. By encoding the data in a way that produces few rising edges, the slow-down caused by the high latency of the rising edge can be further mitigated, increasing the bandwidth of the covert channel.

Additional bandwidth improvements can be made by encoding the data in a way that prevent the magic value from appearing in the data bit stream, removing the possibility of falsely identifying part of the message data as the magic value and thus removing the need for increasing ACK timeouts. We have not exhaustively examined all load configurations. It is thus possible that lower latencies and possibly better resistance could be reached by picking a different load configuration for high-load phases. Future versions of AMD processors might use a different version of Precision Boost 2. Preliminary tests made on a Ryzen 7 3700X processor, presented in Figure 5.7, show a reduced rising edge latency but an increased amount of fluctuations in the resting frequency of different load configurations. This covert channel can possibly be implement with a higher symbol rate on a Ryzen 7 3700X but the high amount of frequency fluctuations and generally lower difference between frequency levels might pose additional challenges.

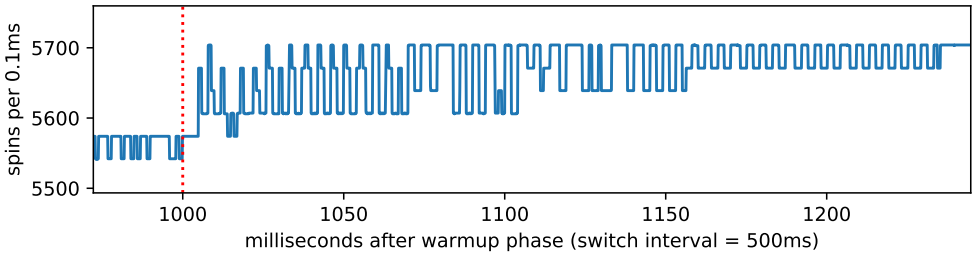
During our tests, we sometimes noticed that the scheduler assigns three busy processes to just two cores. This might be caused by the scheduler preferring to avoid the drawbacks of core switches, such as CPU cache misses. This is a reasonable decision when the active processes are assumed to only be producing load for a short amount of time. In our scenario, however, this causes less cores to be active than intended by the sender, producing either an increased or decreased frequency, depending on whether the overcrowded core is the receiver's core.



(a) Ryzen 7 2700



(b) Ryzen 7 3700X



(c) Ryzen 7 3700X, zoomed in on rising edge

Figure 5.7: Frequency when applying a *low-high-low* load pattern on a Ryzen 7 2700 (a) and Ryzen 7 3700X (b,c). The 3700X has a lower rising edge latency but shows higher fluctuations in frequency and a reduced difference between frequency levels. The dotted lines mark the time at which the load configuration is changed. All graphs include a median filter with window size 1 ms.

Chapter 6

Conclusion

As an additional layer of security, system security policies can demand that applications processing sensitive data be isolated from the internet and other applications. Covert channels provide an attacker with the means to bypass this measure.

Another covert channel utilizing dynamic frequency scaling was recently presented by Kalmbach [8] but relies on specific traits of Intel Turbo Boost. We have presented a covert channel based on AMD's counterpart called Precision Boost 2 that we constructed after an extensive analysis of the frequency changes produced by Precision Boost 2 when modifying the processor load. Our analysis showed key differences between Precision Boost 2 und Turbo Boost, including higher and asymmetric latencies for rising and falling frequencies produced by Precision Boost 2. We accounted for this by implementing latency correction and we have discussed the drawbacks of this approach, specifically the reduced noise resistance. Our covert channel reaches a net bit rate of 1.08 bit/s, more than what is considered "acceptable in most application environments" in the "Orange book" [13]. We have also shown that the same technique could be used to implement a side channel attack on third-party applications, allowing an attacker to gain information on when the application produced load. As the covert channel does not rely on operating system support, constructing centralized software-based countermeasures is not trivial. Because of its low resistance to noise, the current implementation stops working when a third-party applies a high load to at least one core. This could be used as an effective but expensive countermeasure, although it is possible to adjust the covert channel to have a higher resistance against this countermeasure. The comparatively low bandwidth of the covert channel reduces the amount of scenarios in which the covert channel can be used effectively. The implied security vulnerability should, however, still be considered when working with sensitive data of low size or when a system could reasonable be compromised for prolonged periods of time before the intrusion is detected.

The covert channel shown in this thesis could be further improved in the future by adjusting the symbol length to better fit the asymmetry of the latencies. Approaches that use aymmetric symbol lengths and encode the data to avoid the high-latency rising edge could profit from the low-lantecy falling edge and increase the channel bandwidth. The prototype constructed in this thesis also contains long waiting periods to ensure that false-positives when detecting packet headers do not result in a deadlock. An encoding scheme that prevents the magic value in the packet header from appearing in the data bitstream would eliminate the need for this countermeasure.

Bibliography

[1] Hendrik Borghorst. GitHub Repository: rapl-read-ryzen. github.com, 2018. https://github.com/djselbeck/rapl-read-ryzen. [Online; accessed 18-October-2019].

[2] Serdar Cabuk, Carla E. Brodley, and Clay Shields. IP Covert Timing Channels: Design and Detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 178–187, New York, NY, USA, 2004. ACM. http://doi.acm.org/10.1145/1030083.1030108.

[3] Jeffrey Friedman. Tempest: A signal problem. *NSA Cryptologic Spectrum*, 35:26–28, 1972. https://www.nsa.gov/Portals/70/documents/news-features/declassified-documents/cryptologic-spectrum/tempest.pdf.

[4] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299, Cham, 2016. Springer International Publishing.

[5] M. Guri, M. Monitz, and Y. Elovici. USBee: Air-gap covert-channel via electromagnetic emission from USB. In *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, pages 264–268, Dec 2016.

[6] Robert Hallock. Understanding Precision Boost 2 in AMD SenseMI technology. community.amd.com, 2017. https://community.amd.com/community/gaming/blog/2017/11/27/asdasd. [Online; accessed 15-October-2019].

[7] Scharon Harding. What Is an AMD CCX? A Basic Definition. tomshardware.com, 2019. https://www.tomshardware.com/reviews/amd-ccx-definition-cpu-core-explained,6338.html. [Online; accessed 18-October-2019].

[8] Manuel Kalmbach. Frequenz-basierter Covert Channel mithilfe der Intel Turbo Boost Technology. 2019.

[9] S. K. Khatamifard, L. Wang, A. Das, S. Kose, and U. R. Karpuzcu. POWERT Channels: A Novel Class of Covert CommunicationExploiting Power Management Vulnerabilities. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 291–303, Feb 2019.

[10] Gregory Lento. Optimizing Performance with Intel Advanced Vector Extensions. 2014. https://computing.llnl.gov/tutorials/linux_clusters/intelAVXperformanceWhitePaper.pdf.

[11] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[12] P. Miedl, X. He, M. Meyer, D. B. Bartolini, and L. Thiele. Frequency Scaling As a Security Threat on Multicore Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2497–2508, Nov 2018.

[13] United States Department of Defense. Trusted Computer System Evaluation Criteria. page 81, 1985. https://upload.wikimedia.org/wikipedia/commons/a/aa/Trusted_Computer_System_Evaluation_Criteria_DOD_5200.28-STD.pdf.

[14] Srinivas Pandruvada. Running Average Power Limit - RAPL. 01.org, 2014. https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl. [Online; accessed 18-October-2019].

[15] Christof Windeck. AMD Ryzen 2000: Noch bessere Prozessoren. heise.de, 2018. https://www.heise.de/newsticker/meldung/AMD-Ryzen-2000-Noch-bessere-Prozessoren-4027782.html. [Online; accessed 15-October-2019].

[16] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association. https:

//www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.