Can Microkernels Mitigate Microarchitectural Attacks?*

Gunnar Grimsdal[1], Patrik Lundgren[2], Christian Vestlund[3], Felipe Boeira[1], and Mikael Asplund[1][0000−0003−1916−3398]

[1] Department of Computer and Information Science, Linköping University, Sweden
{felipe.boeira,mikael.asplund}@liu.se
[2] Westermo Network Technologies patrik.lundgren@westermo.se
[3] Sectra AB, Linköping, Sweden

Abstract. Microarchitectural attacks such as Meltdown and Spectre have attracted much attention recently. In this paper we study how effective these attacks are on the Genode microkernel framework using three different kernels, Okl4, Nova, and Linux. We try to answer the question whether the strict process separation provided by Genode combined with security-oriented kernels such as Okl4 and Nova can mitigate microarchitectural attacks. We evaluate the attack effectiveness by measuring the throughput of data transfer that violates the security properties of the system. Our results show that the underlying side-channel attack Flush+Reload used in both Meltdown and Spectre, is effective on all investigated platforms. We were also able to achieve high throughput using the Spectre attack, but we were not able to show any effective Meltdown attack on Okl4 or Nova.

Keywords: Genode, Meltdown, Spectre, Flush+Reload, Okl4, Nova

1 Introduction

It used to be the case that general-purpose operating systems were mostly found in desktop computers and servers. However, as IoT devices are becoming increasingly more sophisticated, they tend more and more to require a powerful operating system such as Linux, since otherwise all basic services must be implemented and maintained by the device developers. At the same time, security has become a prime concern both in IoT and in the cloud domain. This is driven both by increasing regulatory demands as well as end-user expectations in this regard. Putting these two trends together we see that operating system security is now more important than ever.

The *principle of least privilege* is a fundamental pillar in security engineering and dictates that any entity should only have access to the information

* Last author is partially supported by RICS: the research centre on Resilient Information and Control Systems (www.rics.se) financed by Swedish Civil Contingencies Agency (MSB) and CENIIT project 14.04

and resources that it needs to fulfil its purpose. In the context of operating systems, this principle supports the use of *microkernels*, or *microvisors* (i.e., minimal hypervisors operating on the same principle). There are many variants of these, but the basic idea is to have as little of the operating system functionalities implemented in the kernel/hypervisor itself. Services such as process and memory management require the CPU to operate in privileged mode and are therefore part of most microkernels, whereas much of the filesystems and many device drivers can be implemented in user mode (given the right system call interfaces). There are today a number of operating system components and framework developed with security in mind. A prominent example is Genode which is a framework for building secure OSs using a microkernel and provides strong isolation guarantees and resource budgeting for individual components. The basic idea is that Genode enforces a recursive and capability based structure, such that components have exact capabilites and may grant any subset of those capabilites to its children. Genode has been developed to run on multiple kernels, such as Nova, Okl4 and Linux.

The security property of such frameworks which guarantees process isolation hinges on basic assumptions on the underlying hardware that have in recent years been shown not to hold. Attacks such as Meltdown and Spectre and later variants thereof rely on CPU optimisations where the processor performs activities that might be useful in future computations, but which are supposed to be invisible to the processes if they are not used. However, by using some side-channel attack (e.g., involving the cache), the microarchitectural state of these tentative computations can leak to the outside.

It would be naive to assume that a microkernel architecture necessarily protects against microarchitectural attacks. On the other hand, strong isolation properties could potentially mitigate some of the proposed attacks by making some or other step in the attack impossible or less powerful. It has been suggested[4] that the impact should be smaller on Genode than on standard OSs, but so far, there has not been any proper scientific studies on this topic. Schmidt et al. [13] demonstrated ways to circumvent security policies for Genode's IPC and implemented a covert channel which abused a file system cache. However, to the best of our knowledge, there has been no previous work demonstrating a violation of Genode's memory separation.

In this paper we ask the question of whether and if so to what extent a microkernel framework together with state-of-the art secure microkernels such as Okl4 and Nova protects against microarchitectural attacks such as Spectre and Meltdown. Building on previous work (often only provided in blogs and discussion forums) we describe how these attacks can be implemented on three different kernels (Okl4, Nova and Linux), all on top of the Genode framework. Since these attacks are inherently based on time-measurements, we discuss how to tune the mechanisms to achieve the highest possible throughput, and also what other measures must be taken to make the attacks work. We demonstrate

[4] N. Feske. *Side-channel attacks(Meltdown, Spectre).* 2018. URL: `https://sourceforge.net/p/genode/mailman/message/36178974/` (visited on 01/16/2019).

that the underlying side-channel attack (Flush+Reload) of both Meltdown and Spectre works well on all three platforms and that same holds for the Spectre V1 attack. For Meltdown on the other hand, while running without problems on Genode+Linux, we have not been able to show a successful attack on Genode+Okl4 or Genode+Nova. We discuss the reasons for this and potential implications.

The contributions of the paper can be summarised as follows.

- Demonstration of how the Flush+Reload side-channel attack and Spectre V1 attack can be successfully performed on Genode using three different kernels.
- An experimental evaluation of the throughput of Flush+Reload and Spectre achieved under different parameter settings.
- Partial results on the effectiveness of the Meltdown attack.

2 Background and Related Work

In this section, we first give a brief introduction to the two main microarchitectural attacks studied in this paper, Meltdown and Spectre, followed by a description of related work.

2.1 Meltdown and Spectre

Meltdown is a microarchitectural attack which exploits the fact that some modern CPUs may execute instructions out of order [9]. Specifically, Meltdown can read memory from an addressable memory space which it should not be able to read from. Lipp et al. [9] used a Meltdown exploit to read memory from the kernel and other user processes in Linux. This was possible as the Linux kernel's memory was mapped into the address space of each user process. Genode's founder Feske has stated that some in-kernel data structures in Genode are likely vulnerable to the Meltdown attack (see footnote 3).

Spectre relies on the fact that some modern CPUs may speculatively execute instructions [6]. There are different versions of the Spectre attack (e.g., [6, 11]), we will be looking at Spectre version 1. Spectre version 1 exploits speculative execution to bypass boundary checks. An attacker could use this attack to execute code which bypasses a boundary check and leaks information to the attacker.

Both Meltdown and Spectre rely on an attacker being able to transmit gathered data to and from the cache. Flush+Reload is a Side-Channel Attack (SCA) which abuses the time difference of fetching uncached and cached data [17]. This channel can be used in the context of Meltdown and Spectre to first read kernel memory into a cache exploiting their respective CPU optimisations. If the address which is cached is carefully crafted, the time with which a process can access this address can be measured to retrieve information.

SCAs extract information from another system or user by abusing some aspects of the system which are not supposed to transmit information. A side channel can also be used as a covert channel, i.e., a channel in which two colluding actors communicate via a side channel.

2.2 Related Work

There has been work on Genode related to security, such as Constable et al. [3] who worked on extending formal Sel4 verification to Virtual Machine Monitor (VMM) running on Genode. Other works have focused on using Genode as a means to achieve a secure OS. Brito et al. [1] used Genode as a secure kernel base to process images securely on an ARM TrustZone cloud environment. However, Genode has seen little work related to microarchitectural attacks and side channels. Schmidt et al. [13] constructed a covert channel in Genode which exploited a software cache to construct a timing channel. However, to the best knowledge of the authors, there has been no other work relating to SCAs in Genode.

Side Channels Xiao et al. [16] demonstrate a covert channel using execution time for write accesses to shared memory pages. They leverage the Copy-On-Write (COW) technique, which is commonly used for shared memory implementations. They also demonstrate, using this technique, examples of a covert channel transmitting 50-90 bps for practical applications.

Pessl et al. [12] present a covert cross CPU channel utilising varying access times of memory banks in DRAM. They demonstrated a channel with a capacity of 2.1 Mbps with an error probability of 1.8% and across VM channel with a capacity of 596 kbps with an error probability of 0.4%

Microarchitectural attacks Mcilroy et al. [11] examined the deep seated implications of how Spectre and incorrect hardware models affect confidentiality-enforcing programming languages. The authors show that that these confidentiality guarantees are completely compromised by Spectre. Koruyeh et al. [7] show that the Return Stack Buffer (RSB) could be exploited instead of the BPU, thus introducing a class of SpectreRSB attacks. Koruyeh et al. were not successful in demonstrating these attacks on ARM and AMD CPUs. However, ARM and AMD CPUs also utilise an RSB and should therefore be vulnerable.

There has also been work examining SCAs targeting ARM Trustzone. Lapid and Wool [8] mounted a side-channel cache attack against the ARM32 AES implementation used by the Keymaster trustlet. Another work by Bukasa et al. [2] demonstrate the ineffectiveness of Trustzone to prevent power analysis SCAs.

Microarchitectural attacks are also a quickly progressing field. A recent work by Schwarz et al. demonstrated the ZombieLoad attack, a new type of microarchitectural attack which exploits a fill buffer to read data from other processes [14]. This fill buffer is a type of load queue which is shared between hyper threads. This buffer can under certain circumstances trigger a load which has been initially issued on another core and thereby can leak data from loads issued by other processes [14].

Security by virtualisation Using a small kernel is not the only way to potentially enhance the security of a system. Another feasible option is to use different virtual systems to separate processes. The virtual systems need to be running on a hypervisor, which may be attacked. Thongthua and Ngamsuriyaroj [15]

discusses some weaknesses they found in popular hypervisor software. However, the abstraction of virtualisation does not prevent microarchitectural attacks such as Meltdown or Spectre [9, 6]. Irazoqui et al. [5] recovered an AES key in a cross-virtual machine setup using a SCA that abused the Last-Level Cache (LLC). The attack is not dependent on the virtual machine running on the same core since the LLC cache was used. Virtualisation also adds to overhead by handling multiple OSs running on the hardware.

3 Methodology Overview

In this section we provide an overview of the methodology used in the paper. First, we elaborate on the problem statement by asking three questions regarding the feasibility of performing microarchitectural attacks on the Genode framework. We then proceed to explain our choices of platforms (i.e., what kernels we investigate) and metric (how the attacks have been evaluated).

3.1 Problem Statement

This paper aims to study the impact of microarchitectural attacks on microkernels. In particular, we investigate effectiveness of Meltdown and Spectre on microkernels. Our investigation can be summarised with the following three research questions.

1. Can Flush+Reload be used to create a covert channel between two processes in Genode, measured as the throughput of demonstrated channel?
2. Are Remote Procedure Call (RPC) mechanisms in the microkernels Nova and Okl4 vulnerable to the Spectre Version 1 (Spectre V1) attack, measured as throughput of demonstrated attack?
3. Can the Meltdown attack be executed on Genode?

We try to answer these questions by implementing these attacks on the Genode framwork using three different kernels as explained below.

3.2 Choice of Platforms

The overall goal of this paper is to study how well a microkernel architecture can withstand the new class of microarchitectural attacks such as Meltdown and Spectre. There is of course a large number of microkernels available and we have opted to study two of them, Okl4 and Nova. Moreover, we decided to use the Genode framework as a common base for both these kernels as well as in combination with Linux. Genode was chosen since it provides the surrounding services needed to run several different microkernels. Moreover, its strict process separation, adherence to a minimal kernel and open-source code nature make it interesting as a basis for secure operating system design.

We chose two microkernels/microvisors Okl4 and Nova that are designed with security in mind and therefore could potentially provide some protection against

the studied attacks. We also tried to use the Sel4 microkernel as it has been formally verified against its specification. Unfortunately, Sel4 on Genode was at the time of our study not well-supported and we did not manage to perform any tests using this kernel.

The Nova kernel, which is a microvisor, is a research project aimed at secure virtualisation. Similar to a microkernel, it provides essential functionality for virtualisation like communication, scheduling and resource management[5].

Okl4 is an open-source microkernel based on the L4 microkernel. It can be used as a hypervisor or as a real-time OS and has been used practically by General Dynamics[6].

3.3 Measuring attack effectiveness

To measure the channel's or the attacks' throughput, a fixed string message m of length n was transmitted. Throughput T was then calculated as the number of correctly transmitted bytes per second (Bps) of transmission. This definition of throughput has been used to measure other microarchitectural attacks [9, 7]. A byte in position i was considered correctly transmitted if the received byte r_i had the same value as the message byte m_i. The throughput of the channel, T, was calculated as

$$T = \frac{\sum_{i=0}^{n} C(m_i, r_i)}{t_n}, \tag{1}$$

where t_n is the total execution time in seconds, and $C(m, r) = 1$ if $m = r$ and 0 otherwise. An array of size 2048 bytes was used to measure throughput. Every leaked byte was forwarded via serial communication to the measuring system.

Genode's timer object was used in Nova and Linux to measure the total execution time, t_n, with millisecond accuracy. The timer object was not used on Okl4; instead, a timer at the measuring system was used to measure t_n. On Okl4, a start-timer command was transmitted via the serial port before the first transmission byte and an end-timer command after the last byte. The timer on the measuring system was started and stopped by these commands. The execution time, t_n was transmitted after transmitting all bytes if Genode's timer object was used.

4 Attack Implementation

In this section, we first describe how the Flush+Reload channel was implemented, followed by a description of the Meltdown and Spectre implementations.

4.1 Implementing the Flush+Reload Channel

We implemented a Flush+Reload channel on all three platforms. Some adaptations were required such as using the rdtsc instruction rather than rdtscp for time measurements on Nova.

[5] *NOVA Microhypervisor*. URL: `http://hypervisor.org/` (visited on 03/19/2019).
[6] `https://gdmissionsystems.com/en/products/secure-mobile/hypervisor`

To setup the Flush+Reload channel, we allocated shared memory to a size of $(256 + 2) * \text{Padding}$. There were 256 addresses to distinguish addresses as different values. These addresses were offset using a padding to prevent prefetching between values (another CPU optimisation). Padding was also used at the beginning and at the end of the array to prevent prefetching of shared memory addresses from accesses outside of the array. By performing memory accesses on this array at a given location the memory location is cached and therefore this is indirectly transmitted. The receiver can then measure access times to each address in the array and conclude which corresponding value was transmitted.

Measuring Cache Hits A threshold was used to decide whether a value was cached or not cached. This threshold was determined by profiling the time it took for the CPU to access cached and uncached values [17]. The Level 1 (L1) cache or LLC was used depending on the attack design. Therefore, two thresholds were defined. One threshold above the L1 cache and one above the LLC.

We assume a memory model of access times as shown in Figure 1. In this figure, t_{LLC} is the upper bound for the LLC and t_{L1} is the upper bound to access the L1 cache. The thresholds t_{LLC} and t_{L1} are chosen as the upper bound of the measurements for the LLC and L1 cache respectively. This choice was made arbitrarily, with the intent of minimizing false positives while preserving true positives.

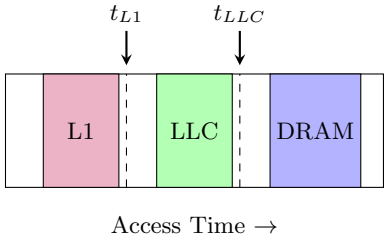


Fig. 1: A model of memory access times for different memory levels.

The time of accessing uncached values was measured by first removing the array from the cache, and then measuring the time for accessing each address. A similar method was used to measure the timings for the L1 cache. Two processes were used to measure the access times to the LLC, one process which cached the values and one process which timed the access time. If the two processes get scheduled on the same core, the values may be cached in either the L1 cache or the LLC.

Preventing Data Prefetching If a program accesses some sequential data from memory, the CPU will prefetch the coming data items to reduce waiting times. However, since the goal of a Flush+Reload channel is to detect cache hits, this prefetching interferes with these measurements. By adding space between the

accessed data items (padding), we can avoid prefecthing. However, too large padding results in excessive memory footprint and slower performance. Therefore, we adopt the use of Strided Read Generator (SRG) [10], which effectively reduces the padding size, while still preventing prefetching. The acces pattern is then $x_{si} = ai + b \bmod m$, where $a \equiv 1 \pmod p$ for all prime factors p in m.

The SRGs were evaluated for the padding sizes 4096, 2048, 1024, 512, 256 and 128. The limits 4096 and 128 were used as they are the page size and cache line size on the tested system. Consequently, the CPU does not prefetch for padding sizes over 4096 bytes and padding below 128 bytes does not guarantee separation between values.

All SRGs where $m = 256$, $a \in [1, 255]$ and $b = 0$ were evaluated. The offset $b = 0$ was chosen as a constant offset should not affect prefetching and to limit the number of SRGs to evaluate. Two SRGs are presented, the one with the best performance in Equation (2) and an arbitrarily chosen worse SRG in Equation (3). The second is used to illustrate the characteristics of a poor performance SRG.

$$x_i = 49i + 0 \bmod 256 \tag{2}$$

$$x_i = 33i + 0 \bmod 256 \tag{3}$$

Reducing Noise To obtain a reliable Flush+Reload channel it may be necessary to make multiple measurements, as done by others [9, 7]. R different measurements, m_{ij}, were taken for any value i with the purpose of increasing the accuracy. A cache hit detection function f_c was used with a threshold of t_c to build a histogram H of recorded cache hits where each entry h_i is the count of detected cache hits for value i. The estimation $\hat{v}$ of the transmitted value v was calculated as $\hat{v} = \max_i h_i$ where,

$$h_i = \sum_{j=0}^{R} f_c(m_{ij})$$

and,

$$f_c(x) = \begin{cases} 1 & \text{if } x < t_c \\ 0 & \text{otherwise} \end{cases}$$

In addition, synchronising was needed to increase the probability of a successful transmission. Locking was used in order to synchronise the transmitter with the receiver.

4.2 Implementing Meltdown

The methodology for Meltdown was based on the proof-of-concept by Lipp et al. [9]. Specifically, Meltdown required methodologies for recovering from a segmentation fault, identifying a target address, obtaining an observable result via a Flush+Reload channel and synchronising the transmitter with the receiver. On the Linux kernel, we disabled the KPTI patch for the attack to work since

the purpose was not to evaluate whether Linux was vulnerable to the attack, but to have it as a baseline implementation.

Recovering from Segmentation Fault Since Genode does not provide support for segmentation fault handlers [4], another method was needed. One possible method is to start a new child process for each read which leads to a segmentation fault [9]. This method allows for transmitting a single byte with each started child. Another method is to use Intel TSX to suppress the fault [9]. Both methods were evaluated, Intel TSX was chosen due to a more straightforward attack design and fewer resource requirements. If Intel TSX is used, no inter-process synchronisation is needed. A process will continue its execution even if non-accessible memory was accessed during a transaction. The attacker can therefore run Flush+Reload directly after the Meltdown attack.

Choosing a Target Address Two target addresses were used, the Linux version banner and a victim process. Previous work has had success with these variants [7] [8]. Furthermore, they were chosen due to the ease of confirming success using an existing working attack.

In the first alternative, the attacker targets a location for a version string defined in the Linux kernel. Confirmation of correct data was done by reading a file using root privileges.

For the second alternative, a victim process was set up to allocate a secret array of 2048 bytes. The array was cached cached by the victim. Thereby, the address and value of the target addresses are known, and the addresses along with its values are cached.

4.3 Implementing Spectre

The design of the Spectre V1 attack consisted of an overall design based on previous work [9] [10]. Specifically methodologies for ensuring speculative execution, training the branch predictor and increasing accuracy by tuning parameters was used.

The attack setup consisted of a victim process and an attacker which shared a common output buffer. The victim was a vulnerable RPC which accessed an array based on an input index and a bounds check, see Listing 1.1. The attacker exploits this by issuing $T_a - 1$ training requests to a victim_function. After $T_a - 1$ requests the attacker issues a malicious request malicious = target_address with an index targeting an address beyond the bounds of the array. For the attack to work, the vulnerable RPC needs to be speculatively executed and the branch predictor needs to be trained.

[7] `https://github.com/paboldin/meltdown-exploit`
[8] `https://github.com/IAIK/meltdown`
[9] `https://gist.github.com/anonymous/99a72c9c1003f8ae0707b4927ec1bd8a`
[10] `https://github.com/crozone/SpectrePoC`

Listing 1.1: Victim Function which is Vulnerable to Spectre V1

```
1 void victim(size_t idx) {
2   if(idx < array_size) {
3     int foo = array[idx]; // May speculatively execute
4     do_something(foo);    // array_size is not in cache
5   }
6 }
```

Ensuring Speculative Execution Speculative execution, according to documents from Intel, is highly dependant on microarchitectural implementation and may vary across different processor families. Kocher et al. [6] state that one trigger for speculative execution is a cache-miss prior to or during branch condition evaluation. Therefore, the boundary check values needs to be removed from the cache. This is done with a heuristic flush of the cache by performing a large amount of memory accesses.

Configuring Variables for Spectre Three parameters are needed to execute the Spectre attack: number of attacks per measurement N_a, the attack period T_a and the number of memory accesses used to flush the cache H_s. Attacks per measurement N_a and T_a were chosen by testing all integers $N_a \in [1, 10]$ and $T_a \in [2, 10]$ to find which combination gave the highest throughput in reading 2048 bytes from the vulnerable process. To determine values for N_a and T_a, H_s was initially chosen to $4096 \cdot 32$, it was then tested using an exponential sample between 64 and the size of the CPU's cache to find a local optimum. It should be noted that the purpose of these local optimisations is not to achieve an optimum, but rather to gauge the possible throughput of this attack.

5 Evaluating Attack Effectiveness

In this section we describe the setup and results of evaluating the effectiveness of the Flush+Reload, Spectre, and Meltdown attacks on the three investigated platforms.

5.1 Setting up System Under Test

The System Under Test (SUT) is composed of Genode with a microkernel core, an attack implementation and an output channel. This setup was executed on an Intel Core i5-7500 CPU.

We used Genode's build tools and documentation to build our implementation for each kernel [11]. These build tools were available at Genode's Github page [12]. To run a build, Genode requires an init-component which is assigned all

[11] `https://genode.org/documentation/developer-resources/index`
[12] `https://github.com/genodelabs/genode/tree/18.11`

system resources. Genode then delegates the task of assigning resources to this init-component. We build our implementation by assigning an initial resource budget to our process, thus enabling it to execute, use RPC and allocate memory. Genode's build tools will from our configuration create files which are used to boot the kernel with our implementation. These files can be used by Grub2 to multi-boot the tested SUT.

5.2 Flush+Reload

Since Flush+Reload is a necessary component both in Meltdown and Spectre, and in itself an interesting subject of study in the context of microkernels we show some results both on how to tune this channel for maximum throughput as well as the final performance achieved.

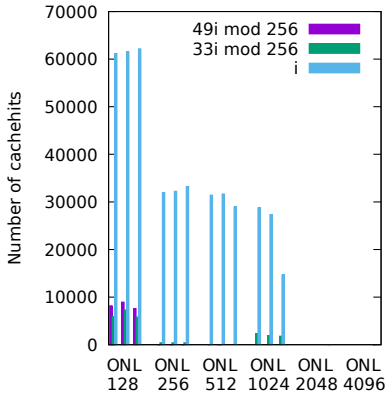
Choosing Cache-Hit Thresholds Table 1 shows the choices of t_{LLC} and t_{L1} for each kernel along with a valid interval for the choices. The valid interval describes the interval in which there are no measurements from the cache level above and all from the desired one. For example, there are no measurements from LLC below 73 cycles on Okl4. Thus, the valid interval for t_{L1} on Okl4 is $[56, 72]$. The choice of t_{LLC} and t_{L1} was chosen as the lowest value in the valid interval.

Kernel	**Chosen t_{LLC}**	**Valid interval for t_{LLC}**	**Chosen t_{L1}**	**Valid interval for t_{L1}**
Okl4	81	$[81, 239]$	56	$[56, 72]$
Nova	80	$[80, 219]$	42	$[42, 64]$
Linux	139	$[139, 239]$	54	$[54, 78]$

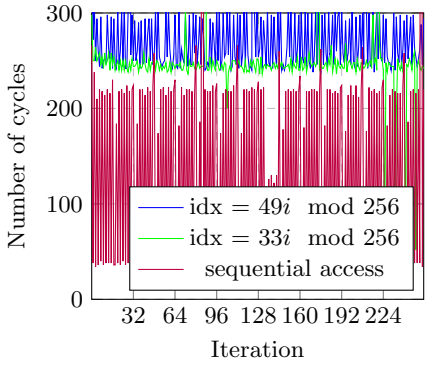
Table 1: The Cache-hit thresholds measured in CPU cycles for each kernel.

Preventing Data Prefetching Figure 2a shows the number of detected cache hits from the array reads using the SRGs from Equations (2) and (3) and different sizes of the internal padding. For each padding size the kernels are denoted using O for Okl4, N for Nova and L for Linux. The SRG $49i \mod 256$ is preventing prefetching at the smallest internal padding and thus results in the smallest memory footprint of the Flush+Reload channel. In Figure 2b, it can also be seen that the SRG $idx = 49i \mod 256$ results in memory access times of almost 300 CPU cycles which is comparable to DRAM access times. Therefore, the SRG in Equation (2) and the internal padding of length 256 was used to obtain further results.

Measuring Throughput Figure 3 shows the throughput of the Flush+Reload channel in six different configurations. For each kernel the transfer was performed within a single process as well as between two different processes. On the x-axis the number of read attempts are shown. Note the logarithmic scale on the x axis.



(a) Number of cache hits from iteration over uncached array using an SRG 256 times on Genode for different internal padding sizes, and different kernels. O = Okl, N = Nova, L = Linux.



(b) Time to access values in a pseudo-randomised or sequential pattern using 256 bytes as internal padding on OKl4.

Clearly, the Flush+Reload channel is effective on all three platforms. In four out of the six configurations the throughput is over 1kBps and sometimes much higher than that. However, when transferring data between two different processes on Okl4 and Nova the throughput is less than 100Bps and reduces to just a few Bps for higher number or read attempts. The most likely explanation we could find for this outcome is due to the way the process synchronisation interferes with the data transfer in these setups. It does not seem to be caused by any attack mitigation mechanism in the kernels.

In all six cases, either a single or two read attempts achieves the best throughput, since while repeating the reads reduces the error rate, it also takes longer time.

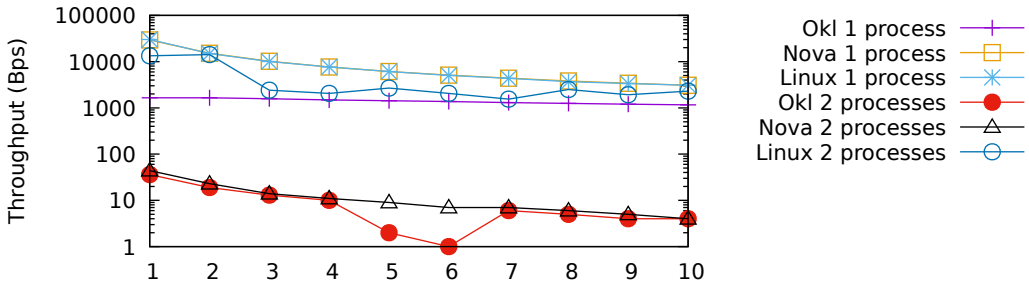


Fig. 3: Reading 2048 bytes within a process or between two processes, for a varying number of attempts

5.3 Meltdown

We were only able to get Meltdown working on the Genode+Linux platform (after disabling the KPTI mitigation). The resulting throughput when read 2048 bytes from another process is shown in Figure 4. The result shows a fluctuating throughput, ranging from 63 to 11070 Bps. This result demonstrates that the Genode framework in itself does not prevent Meltdown (even if we had to adapt the attack as described in Section 4.2)

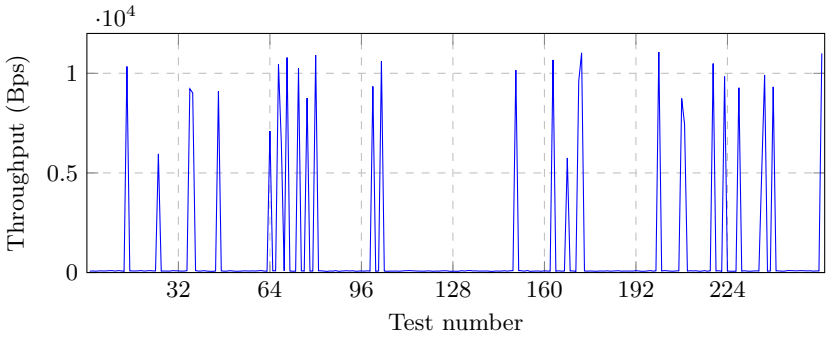


Fig. 4: Throughput from reading 2048 bytes from another process in Genode using Meltdown on Genode+Linux.

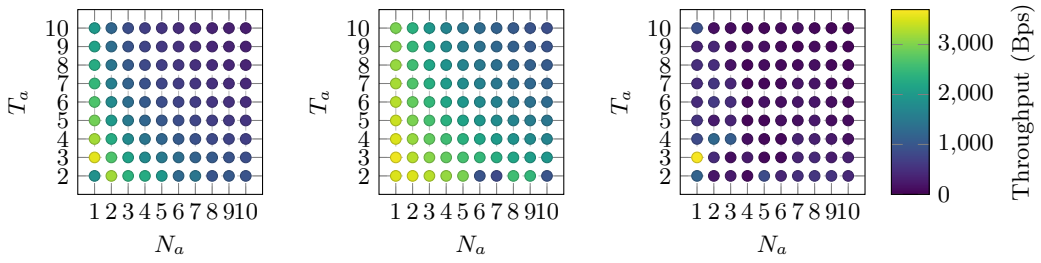
Also interesting is the fact that we were not able to make this attack successful on the Genode+Okl4 or Genode+Nova platforms. Genode makes this attack more difficult to execute by for example not supporting control over which core a process should execute on. However, the main reason we were not able to launch the Meltdown attack on Okl4 or Nova is the lack of mappable address space from another process that can be used in the attack.

5.4 Spectre

We now turn to the Spectre attack. First we show how the parameters were tuned to optimise throughput and then go on to show the resulting throughput.

Attack period, T_a, and number of attacks per measurement, N_a, were tested for $2 \leq T_a \leq 10$ and $1 \leq N_a \leq 10$ on Okl4, Nova and Linux. The result from the tests are shown in Figures 5a to 5c. The results shows that all the kernels have the highest throughput at $N_a = 1$ and $T_a = 3$. Furthermore, the throughput tends to be lower when N_a or T_a approaches higher values.

We also performed an experiment where H_s was varied to find the value that maximised the throughput. The results indicated that the Spectre V1 attack on Okl4, Nova and Linux had its highest through puts at $H_s = 2^{15}$, 2^{17} and 2^{20} respectively. Note that the difference in H_s varies a factor of 2^5 between kernels, thus, choosing a single value for all kernels is likely not suitable.



(a) Throughput out of for different choices of T_a and N_a when reading a total of 2048 bytes on Genode+Okl4.

(b) Throughput of the Spectre attack for different choices of T_a and N_a when reading a total of 2048 bytes on Genode+Nova.

(c) Throughput of the Spectre attack for different choices of T_a and N_a when reading a total of 2048 bytes on Genode+Linux.

The result from trying to read 2048 bytes from an array containing random values with our Spectre V1 implementation is presented in Table 2. The results shows the highest throughput for Nova at 1760 Bps.

Kernel	**Retries**	$\mathbf{N_a}$	$\mathbf{T_a}$	$\mathbf{H_s}$	**Throughput (Bps)**
Okl4	1	1	3	2^{15}	1029
Nova	1	1	3	2^{17}	1760
Linux	2	1	3	2^{20}	525

Table 2: Result of reading 2048 bytes with Spectre V1 with chosen parameters.

Clearly, the Spectre V1 attack is effective on all three platforms and with even better performance on the microkernels compared to Linux.

6 Conclusions

In this paper we have examined the vulnerability of microkernels with respect to the microarchitectural attacks Meltdown and Spectre V1. The targeted microkernels were Okl4, Nova and Linux. These kernels were run within the Genode OS framework for evaluation. Relating back to the problem formulation in Section 3.1 we draw the following conclusions.

- A covert Flush+Reload channel was demonstrated in Genode with a throughput of 36 Bps on Okl4, 44 Bps on Nova and 13409 Bps on Linux. The large discrepancy between Linux and microkernels deemed likely to stem from scheduling differences.
- The investigated microkernels are vulnerable to Spectre V1 and a proof-of-concept was produced with a throughput 1029 Bps on Okl4, 1760 Bps on Nova and 525 Bps on Linux.

– Results regarding microkernels vulnerability to Meltdown are inconclusive. However, an attack reading the secret of another process in Genode running on Linux was demonstrated with a throughput of 11070 Bps.

Clearly, microkernels and Genode are not secure by design against microarchitectural attacks. Microkernels do have some benefits with regards to mitigating Meltdown as several kernels do not map kernel space into user space and are consequently only affected by Meltdown in a limited way. In addition, Genode does not support for custom segmentation fault handlers. Consequently, the Meltdown attack requires another recovery tool, one such viable option is Intel TSX.

One might ask whether these attacks should be dealt with in software at all or if we should simply wait for chip manufacturers to come up with new chip designs. Intel and AMD have announced fixes to some of the known attacks, but at the same time new ones such as Zombieload and Fallout are being discovered. This does not seem to be a problem that will go away by itself. Moreover, given the huge amount of vulnerable processors already out there, often running critical applications, we cannot just sit back and wait. Perhaps future software systems must fundamentally distrust the hardware on which is running, calling for a completely new security model.

For future work, it would be interesting to find an appropriate target for the Meltdown attack against microkernels in Genode and rigorously attack these targeted addresses. It can also be interesting to pursue another segmentation fault recovery design; this is interesting as Intel TSX is only present on some Intel CPUs. With respect to Spectre V1, it may be interesting to target existing Genode components which expose vulnerable RPCs or implement other Spectre variants which use different techniques, such as variants 2, 3 or SpectreRSB [6, 7]. Trying these different variants can further establish the scope of Spectre's impact on microkernels.

References

[1] T. Brito, N. O. Duarte, and N. Santos. "ARM TrustZone for Secure Image Processing on the Cloud". In: *IEEE 35th Symp. on Reliable Distributed Systems Workshops (SRDSW)*. 2016. DOI: 10.1109/SRDSW.2016.17.

[2] S. K. Bukasa, R. Lashermes, H. Le Bouder, J. Lanet, and A. Legay. "How TrustZone Could Be Bypassed: Side-Channel Attacks on a Modern System-on-Chip". In: *Information Security Theory and Practice*. Springer, 2018. DOI: 10.1007/978-3-319-93524-9_6.

[3] S. Constable, A. Sahebolamri, and S. Chapin. *Extending seL4 Integrity to the Genode OS Framework*. Tech. rep. Critical Technologies, 2017.

[4] N. Feske. *Foundations: GENODE Operating System Framework 18.05*. GENODE LABS, 2018. URL: https://genode.org/documentation/genode-foundations-18-05.pdf.

[5] G. Irazoqui, T. Eisenbarth, and B. Sunar. "S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES". In: *IEEE Symposium on Security and Privacy*. 2015. DOI: 10.1109/SP.2015.42.

[6] 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: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019. DOI: 10.1109/SP.2019.00002.

[7] E. M. Koruyeh, K. N Khasawneh, C. Song, and N. Abu-Ghazaleh. "Spectre Returns! Speculation Attacks using the Return Stack Buffer". In: *12th Workshop on Offensive Technologies (WOOT)* (2018), p. 12.

[8] B. Lapid and A. Wool. "Cache-Attacks on the ARM TrustZone Implementations of AES-256 and AES-256-GCM via GPU-Based Analysis". In: *Selected Areas in Cryptography (SAC)*. 2019. DOI: 10.1007/978-3-030-10970-7_11.

[9] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Yuval Yarom, and M. Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium*. 2018. ISBN: 978-1-939133-04-5.

[10] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. "Last-Level Cache Side-Channel Attacks are Practical". In: *IEEE Symposium on Security and Privacy*. May 2015. DOI: 10.1109/SP.2015.43.

[11] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest. *Spectre is here to stay: An analysis of side-channels and speculative execution*. arXiv: 1902.05178. 2019.

[12] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: *25th USENIX Security Symposium*. 2016. ISBN: 978-1-931971-32-4.

[13] W. Schmidt, M. Hanspach, and J. Keller. *A Case Study on Covert Channel Establishment via Software Caches in High-Assurance Computing Systems*. 2015. arXiv: 1508.05228 [cs.CR].

[14] M. Schwarz, M. Lipp, D. Moghimi, J. V. Bulck, J. Stecklina, T. Prescher, and D. Gruss. *ZombieLoad: Cross-Privilege-Boundary Data Sampling*. 2019. arXiv: 1905.05726 [cs.CR].

[15] A. Thongthua and S. Ngamsuriyaroj. "Assessment of Hypervisor Vulnerabilities." In: *International Conference on Cloud Computing Research and Innovations (ICCCRI)* (2016). DOI: 10.1109/ICCCRI.2016.19.

[16] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu. "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation". In: *25th USENIX Security Symposium*. 2016. ISBN: 978-1-931971-32-4.

[17] Y. Yarom and K. Falkner. "Flush+reload: a high resolution, low noise, l3 cache side-channel attack". en. In: *23rd USENIX Security Symposium*. 2014. ISBN: 978-1-931971-15-7.