Rapport

HÖGSKOLAN I SKÖVDE

ONE KERNEL TO RULE THEM ALL

An experimental study inspecting the Meltdown patch effects on the costs of system calls in systemd-nspawn containers

EN KÄRNA ATT HÄRSKA ÖVER DEM ALLA

En experimental studie som inspekterar effekterna av Meltdown uppdateringen i samband med systemd-nspawn containers

Final Year Project in Informatics with Specialization towards Network and System Administration, IT610G, 22,5hp

Ben Kooijman (b15benko@student.his.se)

Supervisor: Johan Zaxmy
Examiner: Rose-Mharie Åhlfeldt

2018-06-03

Abstract

Context. The paradigm of virtualization is rapidly changing due to hardware optimization and capabilities, while also due to rapid development and deployment strategies used in the modern day IT industry. Just like the positive changes, negative effects are necessary to occur in order to improve modern day technologies. This final year project takes a look at both the positive and negatives by exploring how containers are relevant to modern day computing and how they are affected by the patch that mitigates the Meltdown CPU vulnerabilities discovered in mid-2017 in terms of performance. Looking at the trade-off between information security and performance by taking an in-depth approach with a take on the core functionalities of the Linux Kernel. This paper succeeded to identify system call costs that between a secure and non-secure Linux kernel in the context of a containerized environment.

Objectives. This study examines the effects of the KAISER security patch aimed to mitigate microprocessor vulnerabilities related to Meltdown. The investigated effect is the performance as the cost of system calls under the condition of a non-KAISER and a KAISER enabled Linux kernel. The intent is to increase the transparency of how a major security patch such as KAISER affects the system.

Methods. A quantitative experimental study is conducted. One single Debian Stretch node is used with two different treatments. First micro-benchmarks are run without a KAISER enabled kernel which later is compared with a KAISER enabled kernel. The measuring point is the time one single system call takes in a sequence of 1 000 000 system calls.

Results. First macro-benchmarks were conducted to see what a performance loss would look like on an application level. This proved to introduce many superfluous factors which made it difficult to use system calls as a measuring point. In the end a comparison between the two kernels was done. This indicated that the cost per system differed roughly 29% in time.

Conclusions. The results indicate that a large performance loss is identified. However, this does not indicate that all activities on a computer will suffer from this loss. The performance loss the end-user will experience all depends on the amount of system calls generated from one single set of instructions. The performance loss can be neglected if these instructions generating a low amount of system calls. These results should not be used as evidence to favor performance over information security in real life applications and implementations but rather as a motivation to meet the two aspects.

Keywords: Linux, containers, systemd-nspawn, Meltdown, KAISER, system calls

Abstrakt

Kontext. Användning kring virtualisering håller snabbt på att förändras tack vare bättre hårdvaruoptimeringar och förmågor. Samtidigt ändras många olika utplaceringstrategier av datorresurser i den moderna IT industrin. Likväl alla positiva förändringar så introduceras det även effekter med negativa följder. Det här är ett examensarbete som ämnar att utforska båda dessa positiva och negativa effekter. Genom att titta på hur den populära virtualiserings tekniken som består av containers påverkas av uppdateringen som ska lösa sårbarhter i moderna processorer som går under namnet Meltdown. Påverkningarna definieras i form av prestanda vilket tillåter en djupare inspektering av de fundementala funktioner av Linux kärnan, där systemanrop kan användas som mätningspunkt. Metoden samt resultat i det här examensarbetet har med framgång lyckats hitta en kostnads skillnad per systemanrop under förhållandet av en osäker samt en säker Linux kärnna i en miljö som består av containers.

Mål. Examensarbetet tittar på effekterna av uppdateringen som ska åtgärda sårbarheterna i moderna mikroprocessorer relaterade till Meltdown. De investigerade effekterna är definierade som prestandan där kostanden av systemanrop används som mätningspunkt. Meningen med det här examensarbetet att öka transparansen av vad en stor säkerhetsuppdatering gör med ett modernt datorsystem.

Metod. En kvantitativ experimental studie utförs där en Debian Stretch nod används för att observera två olika behandlingar. Till början så körs det ”mikrobenchmarks” under förhållandet av en osäker Linux kärna. Sedan så följs detta upp med en säker Linux kärna. Till slut jämförs de olika resultat med varandra för att identifera hur mycket ett system anrop kostar under en sekvens av 10’000’000 systemanrop.

Resultat. Som pilot studie utfördes det en rad olika tester som använder många olika verktyg för att se om det går att identifiera en prestanda förlust på applikations nivå. Genom att utföra tester utav den karaktären blev det tydligt att överflödiga faktorer förhindrade ett utfall som ansågs vara tillräcklig. Till slut hittades rätt typ av verktyg för att få fram ett utfall som visade att en prestanda skillnad på ~29% existerade mellan en osäker samt en säker Linux kärna.

Sammanfattning. Resultatet indikerar att en prestanda förlust existerar. Dock påverkas inte den generella prestandan nödvändigtvis. Prestanda förlusten en slutanvändare kommer att uppleva beror helt på av hur stort antal systemanrop som genereras under sekvensen av aktiviteten som utförs. Resultatet som redovisas i detta examensarbete bör inte användas som motivation att prioritera prestanda över informations säkerhet i produktions miljöer men hellre som en motivation att kunna bemöta båda aspekterna.

Nyckelord: Linux, containers, systemd-nspawn, Meltdown, KAISER, systemanrop

Table of Contents

1 Introduction....1

2 Background....2

2.1 Containers....2

2.2 Systemd-nspawn....3

2.3 Container components....4

2.3.1 Namespaces....4

2.3.2 Cgroups....7

2.4 System security....8

2.4.1 LSM....8

2.4.2 MAC....9

2.5 Vulnerability and mitigation....9

2.5.1 Meltdown....9

2.5.2 KAISER....11

3 Problem formulation....13

3.1 Research question....13

3.2 Related work....14

3.3 Ethical aspects....14

4 Method....14

4.1 Experimental design....15

4.1.1 Variables....15

4.1.2 Treatments....16

4.2 Benchmarking....16

4.3 Systemd-nspawn container....16

4.5 Used Tools....17

4.6 Threats to validity....18

5 Results....19
5.1 Pilot experiment....19
5.2 Main experiment....20
5.2.1 Main experiment - Results....20
6 Conclusion....21
7 Discussion....22
7.1 Experimental process....22
7.2 Limitations....23
7.3 Reflections....23
8 Future work....24
References
Appendix A – Systemd-nspawn configuration
Appendix B – Scripts
Appendix C – Compiling a Linux kernel

1 Introduction

Virtualization is a mature domain among one of the most common computer use-cases today. Even though these technologies hide behind the scene of modern day computer usage, chances are that most everyday computer usage by end-users has some involvement of virtualization one way or another. In most cases it consists of running multiple instances of virtual environments on a single physical host in order to make use of the available resources on a set of hardware. In later years the concept of virtualizing has become more attractive than ever, which makes it a competitive area in terms of technology development and choices of deployment strategies for companies of varying sizes. This is because of how much hardware has increasingly become better throughout the years. Using physical hosts and the set of hardware available, virtual environments make sense in terms of utilizing resources, ultimately changing the ratio between monetary costs and hardware utilization in favor of the latter. In addition, other motivations such as hardware independence, isolation, scalability and information security reasons in terms of user environments and diverse properties optimized for different use cases are considered as well (Morabito et al., 2015).

Due to the recent events of the Spectre and Meltdown vulnerabilities found in modern day CPUs, many questions have been raised. This work aims to focus on the latter in relation to the patch that mitigates the related problems introduced by Meltdown. This mitigation refers to the KAISER patch which introduces interesting new characteristics to the Linux Kernel, which potentially affects the advantages of OS-level virtualization in terms of performance in a negative way.

A background is given in chapter two. Here information about containers as a concept and their implementation on a technical level is given, followed by explanations of Meltdown and KAISER. Then in the third chapter, the problem description is formed, this covers earlier research, ethical aspects and the question for this final year project's topic. Further on the fourth chapter mentions the method and experimental design which are used to answer the research question. The results of the experiments are presented in chapter five and are then followed up with discussions, conclusions and reflections which are presented in the sixth chapter. This work is then finalized with future work in chapter seven.

2 Background

This chapter presents the concepts of containers, systemd-nspawn, The Linux Kernel, Meltdown, and KAISER.

2.1 Containers

Containers are often compared with bare-metal virtual machines and fall under the paradigm of OS-level virtualization/paravirtualization. Containers are virtual instances (guests) of applications running on top of a host operating system. The main difference between OS-level virtualization and Hardware-level virtualization is the kernel interaction on both the host and the guest. With hardware-level virtualization, each guest is a virtual instance of a complete computer containing an OS with its own kernel interacting with emulated or directly allocated hardware on top of a Hypervisor system. In contrast, OS-level virtualization works by sharing the host's kernel between guests, this is seen as a layered approach in Figure 1 (Morabito et al., 2015).

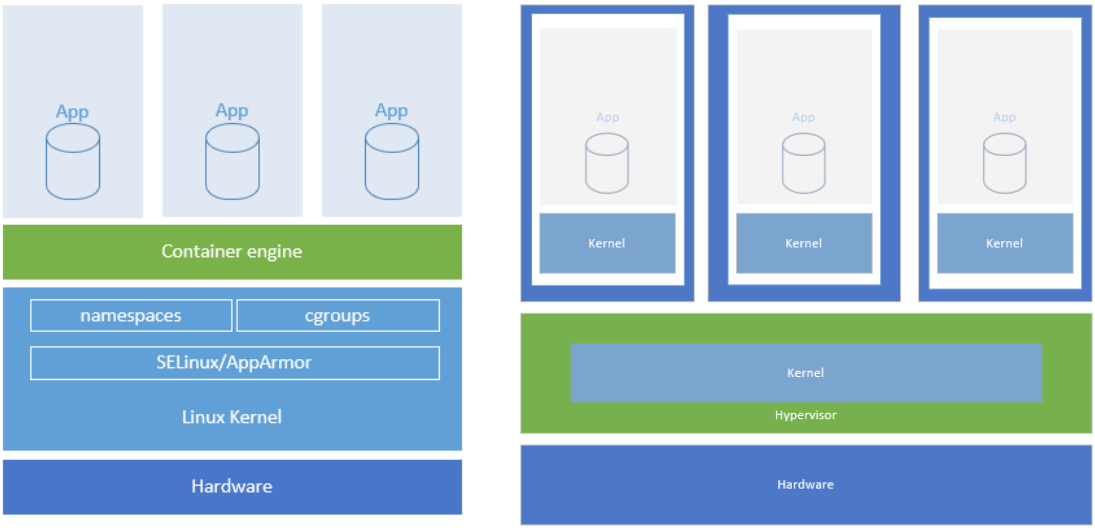

Figure 1 – OS-level virtualization and hardware-level type 2 virtualization.

Figure 1 on the right represents type 1 bare-metal virtualization. This is recognized by the hypervisor running directly on the hardware. The important difference between the right and the left is the location of the kernel. Each guest operating system inside a virtual machine has its own kernel isolated from other virtual machines/instances and the host on the right. While only one kernel exists no matter the number of containers seen on the left side.

These containers often contain a minimal amount of resources needed to run the applications inside of them. A container engine handles how these applications interact with the underlying operating system by acting as a link between each container and the kernel. Because of this kernel sharing, no hardware-emulation is necessary. Because of this increased speed and efficiency in terms of start-up time, the density of containers per physical host, availability and hardware independence become true (Grattafiori, 2016). Even though containers introduce many advantages, some limitations exist. Due to the host kernel being shared between each guest they must be compliant with the kernel, ultimately, meaning that the guest is limited by the host's functionality. The kernel sharing also introduces some information security considerations since isolation is weakened compared to hardware-virtualization. As Grattafiori (2016) explains, the shared resources create a larger attack surface

compared to hardware or full virtualization which can comprise the host and the guests that it is serving. This is because vulnerabilities which can be exploited by fundamental computing characteristics such as system calls, networking, disk access and the CPU are all introduced by the applications run inside of each container. Meaning that the sufficiency of isolation is defined by the host's ability to prevent comprises which can lead to information leakage originating from one or more containers. This paradigm of virtualization has over the span of recent years become relevant due to its many use cases. The first reason may apply to all virtualization types, which is technical information security; In server environments, many different services are running constantly. Each service contributes to an attack surface for a domain. If one of these services is tampered with by an intruder, the comprised service may act as a gateway to other services. Isolation is therefore a preventative measure countering this vulnerability.

Other reasons are more specific to OS-level/paravirtualization; Many companies and organizations have been implementing containers in their infrastructures, this process of paradigm-shifting is also referred to as containerization, where former single-purpose fully virtualized machines on hypervisors are replaced by containers. Full virtualization provide the best isolation and resource distribution mechanisms. However, containers are efficient at offering the most out of a set of hardware if a lot of small isolated processes have to be run. This is because of the lightweight advantages mentioned earlier while also achieving a degree of isolation. This is attractive for companies such as Google, Amazon, and Microsoft who own large data centers which offer cloud platform solutions to their customers. Here the ease of deployment allows a lot of flexibility for system administrators and developers to deploy both production and testing environments. In turn the low resource cost offer less power consumption which also benefits the providers (Scheepers, 2014).

For further reference, the terms process, guest, namespace members, and containers are intertwined for contextual reasons.

2.2 Systemd-nspawn

Systemd is the service handler or the current ”init system” in modern Linux distributions. It has its own set of components such as system utilities, daemons, targets, core and libraries on top of the Linux kernel. All of them are aimed to contribute to the fundamental functionalities of Linux systems. Each component brings its own category of functionality, where for example the daemons handle service logging and service handling, targets handle run levels, and the core among many things brings features such as mounting, swapping and socket creation. Interestingly among all features, systemd also introduces a container functionality which separates itself from the already existing Linux Containers (LXC) and libvert software. This is systemd-nspawn. One of the creators Lennart Poettering explained in 2013 that systemd-nspawn originally was a tool for debugging in development environments and told that it is like the chroot command on steroids (The Linux Foundation, 2013). Chroot is not a containerization technology but rather a tool which allows a user to change the location of a root directory residing on a Linux-based system. It needs no necessary configuration and is overall a simple tool. When systemd was under development the project somehow needed to manipulate how root directories worked, but more complexity than chroot was necessary. This was the introduction of systemd-nspawn. Some functionalities such as namespaces and control groups (cgroups) were added for resource management, and unlike chroot, systemd-nspawn does need some configuration. However, the configuration is so minimal it comes close to chroot and ultimately provides the same functionality at a minimum. When creating a systemd-nspawn instance, systemd is the actual part which isolates resources whereas systemd-nspawn practically only manages those isolated resources. It basically works by first creating an isolated space and then launching a

Linux-based operating system inside of it. It is not meant to run in production environments such as LXC or Docker, which is seen by the non-existent image repository and a non-standardized file extension of the used images. However, this is not a drawback since systemd-nspawn rather aims at "building, testing, debugging and profiling" which only benefits from these flexible characteristics (Edge, 2013). Because systemd-nspawn is a feature of systemd, it is a feature close to the Linux kernel and supports many of the features mentioned earlier. It is even possible to run other container systems such as Docker with systemd-nspawn as the container engine, therefore making it a good technology to rely on due to the simplistic low-level characteristics.

2.3 Container components

The kernel in Linux is not restricted to containers. However, in contrary containers are restricted to the kernel and its characteristics. This is because of the non-hardware emulating aspect of containers. No matter what container engine is used, some major underlying components are always found. These components are seen inside the kernel and provide characteristics such as; non-hardware emulating operation, application isolation with namespaces, hardware resource limitations with cgroups, root capabilities that regulate the power of the host root user in respect to its guests, interaction with container environments with different root filesystems, and mandatory access controls (MAC) with SELinux or AppArmor.

2.3.1 Namespaces

Namespaces are the building block for isolation and separation in computer processes. The usage of namespaces allows containers to create their own view of the system. As mentioned earlier, chroot allows a user to change the path of a root directory. Namespaces take this concept of chroot further by allowing more fundamental operating system functionalities such as networking and process trees to be moved to other paths as well. This allows a container to act more like a virtual machine and less than a filesystem mounted on an arbitrary directory.

There are several types of namespaces, each purposely utilizing a type of resource found in the Linux operating system. These types are; Mount, Unix Time-sharing System (UTS), Inter Process Communication (IPC), Network and User. The reason why namespaces are such a severe building block of the container system anatomy is due to the ability to isolate the resources used by processes, ultimately, limiting access to the host's resources for each guest container without them being aware of these limitations. How this is observed by the host operating system and an active container is illustrated in figure 2.

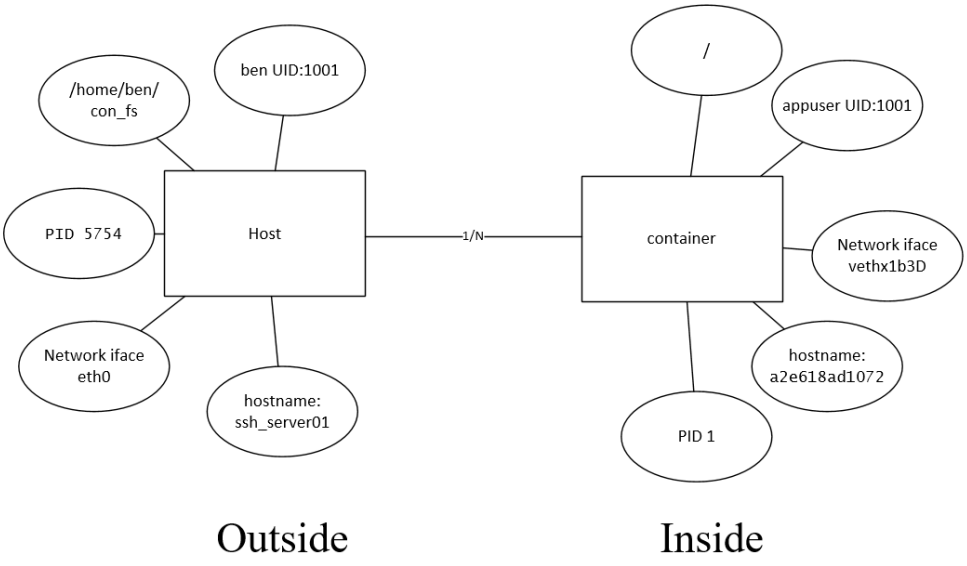

Figure 2 – Namespaces

The outside describes the host system on which the container resides on. The root directory on the inside is actually a directory named “con_fs” on the outside which resides inside the host’s home directory belonging to the user “ben”. Notice the UID of both sides, the names are different even though the UID value is the same.

The namespace implementation in Linux allows for example an unprivileged processes executed by a normal user, to have the ability to create user namespaces in which the privileges are unlimited. In turn, this process can create another namespace type within the user namespace. This makes it possible for the normally unprivileged processes to execute privileged commands within its own user namespace inside the container. For example, the system file such as "/etc/shadow" has completely different permissions inside the container when compared to outside the container or to other containers (Kerrisk, 2013). Each namespace type is further explained in this section.

- **Mount namespaces**

  Members of a mount namespace have a different view of the filesystem hierarchy when compared to the host. For instance, the outside host has its own ext4 filesystem with several directories in different places. Each one of these directories may act as filesystems for members of a mount namespace. If one of those directories were to be used as a mounting path for a container, all common directories such as ”/etc/”, ”/var”, ”/home” etc. would be created within that directory. The member of the namespace would also not have the knowledge to know that anything else exists outside of that particular directory.

  This namespace also ensures that system calls such as ”mount” or ”umount” are unique to each filesystem hierarchy. For example; a mount operation within a container does not affect the mount point in the outside host’s filesystem (Kerrisk, 2013).

The disk size of the namespace mount is determined by the amount of disk space allocated towards the directory on the outside host system which the mount namespace uses.

- **UTS namespaces**

  The name derives from a parameter used for the involved ”uname” system calls; ”Unix Time-sharing System”. Containers act as individual hosts, meaning, that they also make use of host and domain names. In order for each container to have a unique fully qualified domain name (FQDN), hostname and Network Information System (NIS), they must be a member of a UTS namespace.

  The namespace isolates node name and domain name which are system identifiers for the aforementioned hostname and domain conventions. These are returned by the ”uname” system call which according to the uname manual (man) page in Linux can be used to gather kernel information. As described earlier, containers share it’s host’s kernel, meaning that the information gathered from the kernel is changed by UTS namespaces with the purpose of creating individual containers uniquely recognizable by name on a network (Kerrisk, 2013).

- **IPC namespaces**

  IPC stands for Inter-Process Communication which is a collection of varying technologies used for internal communication between threads and processes. This is similar to a client/server model where the client requests data and the server responds to the client in order to serve data towards the client. Depending on the context of data requesting/responding, processes can be either the client or the server.

  An example can be; if a user copies a text string from one file into another. The clipboard on a computer holds this data and acts as a data sharing depository among the services running. This is a basic example of IPC (Microsoft, 2018).

  IPC namespaces use these technologies in the same manner as traditional IPC in non-container systems. According to the Linux man pages for namespaces, System V and POSIX message queues; IPC namespaces isolate resources such as System V/UNIX IPC objects and POSIX message queues. These System V/UNIX IPC resources are one of the following fundamental operating system management implementations such as message queues, semaphores, and shared memory segments. While POSIX message queues simply are queues used by processes to exchange data in the form of messages. The important thing which makes IPC namespaces viable for containers is the fact that all resources used by an IPC namespace are only seen by processes that are members of the IPC namespace allocated towards a container. One member may act as the data depository while another gathers data from it.

- **Network namespaces**

  Network namespaces are utilized in order to create network communication between containers, the host system, and external hosts. For the container' point of view, each network namespace creates a loopback and network interface to which an IP address can be assigned to. These network interfaces are virtual and often bridged to the physical network interface of the host system. The result thereof is the ability to bind multiple inet sockets to one or several containers. For example, multiple containers on the same host system can be listening on port 443 for incoming HTTPS requests (Marmol, Jnagal & Hockin, 2015).

- **User namespaces**

  Each user on Linux systems has user identifiers (UID) and group identifiers (GID) which are used by the operating system to determine if users are privileged or not. If they are, the users can perform a task requiring elevated permissions. Usually, a normal user cannot perform any elevated task if they have not been granted permission to do so. By default the UID/GID 0 is used by root which can perform all tasks no matter what permissions are required.

  User namespaces isolate these identifiers. Meaning that processes (containers) can have a UID/GID of an unprivileged user outside the namespace but a privileged user with UID/GID 0 inside the namespace (Kerrisk, 2013).

2.3.2 Cgroups

Control groups (cgroups) are a fundamental component used for managing and monitoring system resource allocation toward containers. Conceptually these cgroups are similar to ”nice” which is a command used to regulate the priority of a running process in Linux systems and thereby controlling resources depending on the value of the ”nice level”. However, in practice cgroups allow more sophisticated management. The definition of system resources are CPU time, primary memory allocation, secondary memory allocation, networking, and monitoring. Managing resource allocations is done by creating groups which are associated with tasks (tasks are system processes in cgroup terminology). Each group is a collection of resource specifications used to control how much a task is allowed or not allowed to have. Normally the Linux operating system already does this dynamically, but in some cases, it is necessary to guarantee a specific amount of resources to a task. For example; when running several container tasks, it should not be possible for one particular container to consume too much primary memory, eventually killing its neighbouring processes if cgroups are implemented correctly. It is even possible to deny resources to a task if this is necessary with the use of cgroups. For each task, it is possible to monitor the consumption of resources defined in a cgroup which can be used to dynamically adjust resources on the fly. In the context of containers, the outside/inside analogy mentioned in chapter 2.3.1 figure 2 is relevant even in the case of cgroups since each container is not aware of its given cgroup because the resource allocation takes place outside of the container. A visual example of how it all works is seen in figure 3.

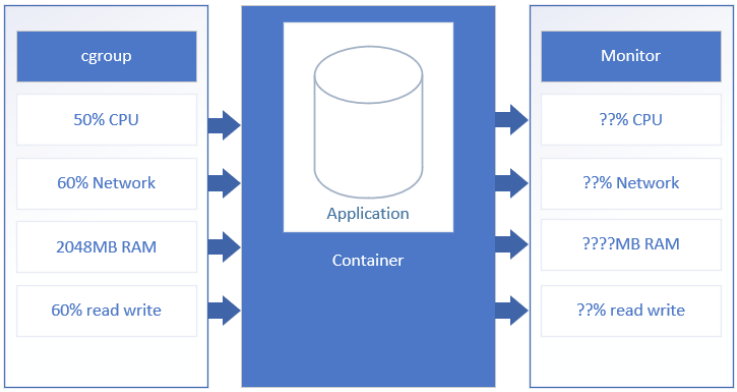

Figure 3 – the left pane specifies the number of resources which are controlled by the cgroup and allocated towards the container. The most right pane then monitors how much of those resources actually are used by the container.

Similar to the hierarchical Linux process model which consists of one parent (init) from whom all other processes inherit from, cgroups also share a hierarchical approach to task structures. The main difference is that

in the traditional Linux process model only one tree exists due to the fact that the absolute path to all processes begin with the init process. While many different cgroup hierarchies can exist with parent and child cgroups who can be used by tasks (Redhat, 2018). This is seen in figure 4.

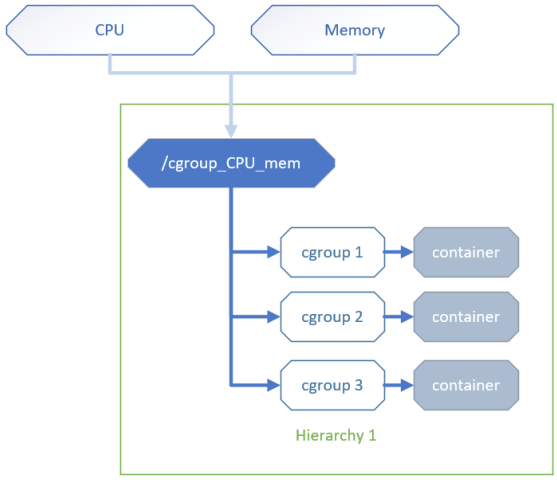

Figure 4 – Both CPU and memory resources are specified in a cgroup file which is created by the Linux kernel. The cgroups in the picture represent specifications of what tasks are using the specified resources, where the tasks are the shown containers.

2.4 System security

Like many problem areas in which information security is a relevant, a single solution often is not the only solution to a given problem. Many times a layered approach is needed. The definition of how many layers, and what each layer does varies depending on who is asked and what context looks like. When it comes to virtualization, especially containers compared to fully virtualized machines, their purposes from an administrative standpoint are generally different even though the outcome of their usage from a consumer perspective is similar. Virtual machines introduce a better information security model because of the strong isolation. As mentioned earlier isolation provides protection from the host and the residing guests from being comprised. However, the trade-off between information security and performance is larger when using fully virtualized machines. This does not mean that containers are less suitable for production usage. It mainly comes down to a variation of best practices such as hardening host and guests and not using multiple applications in one container. In addition this is enforced by combining those practices with the use of technical implementations such as Linux Security Modules (LSM) and Mandatory Access Control (MAC) (Raj MP et al., 2016). By default, containers already achieve a level of technical information security due to the isolation characteristics of namespaces and cgroups, this is hardened with dedicated mechanisms such as LSM and MAC.

2.4.1 LSM

LSM is not designed for containers per se, but containers are affected by its presence. LSM is a framework that provides a modular approach to implementing so-called ”security policies” such as SELinux or AppArmor in the Linux operating system. By saying that containers are affected by LSM is indirectly accurate. In reality the security policies defined by MAC determine the outcome of how a container behaves. But without LSM the MAC security policies cannot be present. Core characteristics of LSM allow a modular approach to user authorization on a Linux system. Instead of implementing one single security model which everyone has to use, a user can simply detach and attach different security models. The LSM framework works in the following way: When a unprivileged user process executes a system call, it first traverses through the Linux Kernel which then, in turn, will check with the LSM framework and asks if this system call is permitted. The LSM relays the

question to the posing installed security policy. The LSM will answer either "yes" or "no" depending on what the security policy says (Wright et al., 2002).

2.4.2 MAC

As in most operating systems, Linux included. The security model used is based upon permissions for the users. Without any MAC configured, directories, files, and processes are owned by users. The actions which can be performed with them are determined by the permissions set on that particular user in relation to the directory, file or process. This is also known as Discretionary Access Control (DAC). However, in Linux the root account bypasses DAC, ultimately, making everything possible with this user account. While this is great for comfort reasons and for information security to some extent if the root user is used properly, it introduces a concern as well. If an attacker manages to get root access to a system, then there are no real limitations to what is possible. Due to the flexibility of LSM, it is possible to change this behavior and implement MAC security models. These models basically eliminate the need for a root account since it introduces security policies which are set by the owner of the system. Kerr & Alves-Foss (2016) explain that MAC is based around objects and subjects, which represent data and users respectively. Access to objects is controlled by looking at the relationship between the object classification and the subject's clearance. Implementations of such relations can be seen as policies. Once these policies are in place, they cannot be overwritten by another subject, not even if it were to be root (Turnbull, 2006). SELinux and AppArmor are two MAC security models widely used in many Linux distributions and containers are in turn affected by them. If for example a container runs as root and is comprised because it has been a target of a network attack, the underlying directories, files and processes are protected by MAC because their permissions cannot be overwritten by the comprised container. As the Redhat documentation says: "MAC enabled systems limit the scope of potential damage that can result from the exploitation of vulnerabilities in application processes and services" (Redhat, 2018).

2.5 Vulnerability and mitigation

Around the spring of 2017 a list of vulnerabilities were found in the modern microprocessors. These processors are the most common type of processors in the world and make sure that everything from mobile devices to large data center servers fulfill our everyday computing needs. Because of this wide area of effect, this topic has been in the scope of view ever since, and 2018 promises many workarounds and designs of improved CPU architectures. The vulnerabilities found are named Meltdown & Spectre which both exploit the way processors manages memory referencing and control flow optimization. Lipp et al. (2017) are the authors of the whitepaper concerning Meltdown (CVE-2017-5754) and explicitly mention how containers are affected by these vulnerabilities, while Gruss et al. (2017) explain a potential solution named KAISER. This chapter will cover and discuss important characteristics concerning both Meltdown and KAISER with respect to containers.

2.5.1 Meltdown

Containers allow processes to be isolated from an underlying operating system. However, memory management has no exception for containers. Each containerized process uses memory addresses just like any process outside a container. Due to the fact that the Meltdown vulnerability is found inside the processor on a lower-level close to the hardware, it is agnostic to operating system characteristics. The characteristics of application-level processes such as containers are completely ignored if this vulnerability is exploited. Lipp et al. (2017) explain

how modern microprocessors operate with the onboard memory in order to achieve isolation between processes while minimizing the cost of context switching. This is achieved by an additional bit set by the processor known as a supervisor bit. This bit determines whether a kernel memory page can be accessed or not by a particular process. This can be seen as a simple authorization mechanism that manages processes for whether or not they can see other processes.

The kernel space is the upper part of the physical memory which a computer uses to store the code of the kernel. This space exists due to the fact that processes running under user space are unprivileged and only have access to a limited part of the physical memory. Whereas the kernel space contains code used by the operating system and must therefore have access to all of the physical memory. The purpose of this supervisor bit is to allow certain parts of the kernel space to be mapped towards unprivileged processes while maintaining the integrity of those unprivileged processes at the same time. The integrity is kept because unprivileged processes cannot see another process that is tagged with a supervisor bit. Such a supervisor bit tag occurs when kernel space is entered and cleared when switching back from kernel space to user space. This allows unprivileged processes to request system functionalities found in the kernel space by using system calls. For example; unprivileged process #1 wants to enter kernel space, and while there, unprivileged processes #2 cannot see unprivileged process #1 due to the set supervisor bit. Figure 5 shows a visual representation user and kernel space.

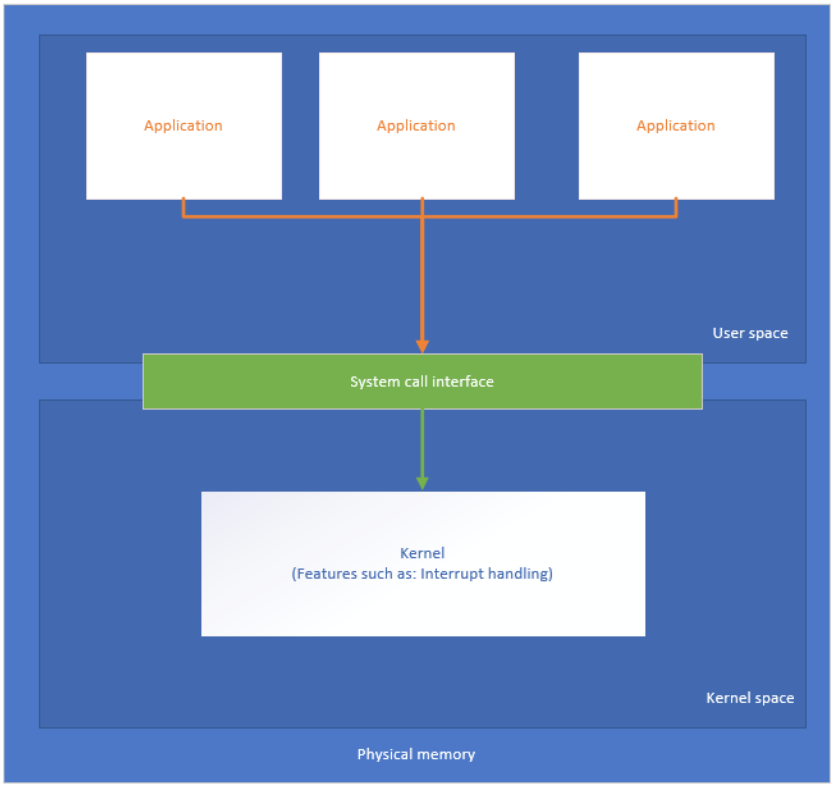

Figure 5 – User space and kernel space.

Meltdown bypasses the user space and kernel space isolation which allows any user process to read the entire kernel space, successfully gaining access to read the entire physical memory by exploiting ”out-of-order-execution”. Out-of-order-execution is a performance feature found in many modern CPU’s today. This feature

aims to reduce the time it takes (latency) for the CPU to fetch information from the physical memory by speculating which processes will need information in the near future. If such a process is identified, the processor interprets this information and starts completing the instructions before they are needed. Many modern processors have four cores. Imagine core 0 is utilized at 100% while the other three remaining cores are idling. By utilizing idling cores, the processor can plan ahead of time. Which makes it possible to fetch the results of future instructions, instead of calculating them when they show next in line inside the CPU scheduler's control flow. The outcome of out-of-order-executions is stored in caching locations. In order to understand what the Meltdown vulnerability refers to, it is important to know how user-space processes are executed without out-of-order execution. Normally when an exception is raised during runtime for any reason at all in a user-space process, it is trapped to the kernel and continues inside kernel space. This means that other user-space processes cannot see the trapped process since they lack privileges defined by supervisor bits telling that the control flow is continuing in kernel space (privileged mode). But imagine this same process again, but this time it has been executed out-of-order, meaning that the CPU has looked at the needed instructions ahead of time, ensuring that there is no dependency on the aforementioned exception. Supervisor bits are in this case not used for memory pages since they originate from out-of-order execution instructions. Side channel attacks can exploit this, by looking at one factor in order to gain information about another factor. By looking at the cache, only containing a limited amount of memory references per look-up, an attacker can successively gain information about the parts of the kernel-space layout and with given time, the entire kernel space. Meltdown attacks can exploit this vulnerability by making the CPU perform out-of-order-executions and transferring cache contents consisting of memory references without supervisor bits to a state where it can be further processed. Normally the common state of memory references in the cache is not "processable" by unprivileged processes since they are part of CPU instructions. A successful Meltdown exploitation from a side channel attack eliminates the boundary between privileged and unprivileged authentication defined by supervisor bits.

In the context of Linux containers, Lipp et al (2017) explain that Linux kernel versions 2.6.32 to 4.13.0 (which is a timespan of over ten years when looking at the large files mirrors used by many Linux distributions (Kernel.org, 2017)) have been tested and showed results where Meltdown succeeded to gain full kernel space access. Meaning that the entire physical memory was exposed as well since the kernel space in Linux contains information about the whole physical memory. In addition to this, Docker, LXC as well as OpenVZ were tested and proved it is possible to mount a Meltdown attack inside a container in order to gain information about other container guests and the underlying host operating system.

2.5.2 KAISER

As of today, there are no standard mitigations which tackle the problem at the same layer of the vulnerabilities. This is due to the difficulty of patching hardware. However, the Meltdown whitepaper mentioned one temporary solution going under the name of KAISER. This mitigation is not directly connected to the discovery of Meltdown but its intentions are well fitting even though it is a software solution. The recent version of the Linux kernel includes KAISER. In the Meltdown whitepaper, the authors describe a information security to performance trade-off which knowingly exists, but is not explained in-depth and is advised as future research.

The authors Gruss et al. (2017) motivate this by telling bugs can be found in both user and kernel space, meaning that vulnerabilities exist no matter the privileged and unprivileged boundaries of the virtual memory in modern systems. Characteristics of modern computer systems such as the supervisor bits mentioned earlier make

it harder to exploit kernel bugs but not impossible. This is justified by earlier research that discusses features such as; Kernel Address Space Layout Randomization (KASLR), Supervisor Mode Access Prevention/Supervisor Mode Execution Prevention (SMAP/SMEP) and the exploitation of them. In order to see how KAISER is relevant, those features must be explained. The intention with KAISER is to prevent knowledge about virtual-to-physical memory mapping to each user space process. Because those mappings make it possible to bypass the aforementioned fundamental security features included in modern x86/x86-64 CPU architectures. KASLR randomizes the system's virtual memory layout during boot time or on process creation (Hund et al. 2013). This makes it hard to find patterns of different address locations to an attacker. Also, SMAP/SMEP are security mechanisms that provide prevention of direct access to kernel space when user space is active. Hund et al. (2013) describes the limitations of KASLR and also presents attacks exploiting double page faults which evade KASLR boundaries, giving knowledge about kernel address space layout to an unprivileged process. Another attack takes this further by exploiting out-of-order instructions. This introduces attacks that allow unprivileged processes to obtain address information which circumvents the SMAP/SMEP and KASLR boundaries (Gruss et al., 2016). The last attack by Jang et al. (2016) uses characteristics seen in a CPU feature named Intel TSX in order to exploit KASLR resulting in a de-randomization of address space layouts. If those scenarios were to happen in the context of container usage, the isolation between each container and its host would not matter to an attacker due to the commonalities of all those attacks. Which can be summarized as hardware level vulnerabilities. This makes it a cumbersome task to mitigate the vulnerabilities without changing the design of the hardware itself. The purpose behind KAISER is to change the behavior of how modern x86/x86-64 processors utilize memory in order to harden the isolation between user space and kernel space, making the mentioned attacks obsolete.

KAISER has three distinct characteristics which originate from mitigating challenges and limitations which arose from hardening the isolation between user space and kernel space. The first is the use of shadow address spaces. Two address spaces for each process are created. One that is mapped to the user space with no presence of the kernel. And another user space mapping including the kernel as well with a SMAP/SMEP protection. SMAP countermeasures invalid user space memory references and SMEP prevent user processes code executions inside the kernel. The idea of shadow address spaces is to co-operate with the high level of parallelism used in modern processors. Parallelism is a characteristic where threads can be active concurrently on one single system. Threads cannot use the same page table structures in user space and kernel space. This would result in a huge amount of synchronization overhead because one single thread can affect the operation of other concurrent threads. Registers inside the processor are keeping track of both those address spaces, these registers are named CR3 registers. The second characteristic focuses on multiprocessing. The x86/x86-64 processor architecture is heavily based on interrupts and context switches which allow threads to run on multiple cores. This ultimately means that some kernel addresses must reside in user space in order to allow multiprocessing due to nature of how interruptions work. However, this proves that the presence of kernel space addresses inside the user space cannot be eliminated to a full extent with KAISER. The third focuses on preventing a huge performance loss due to Translation Lookaside Buffer (TLB) flushes. A TLB is a special cache located inside the processor to speed up address translation and privilege checks (Gruss et al., 2016). It holds entries just like a page table, mapping virtual to physical addresses. Meaning that if it were to be empty, the time for memory management would increase. Such a flush can occur when switching completely between a address space (user space to kernel space or vice versa, this is referred to as mode switching). This is because CR3 registers must be updated on each context or mode switch, which acts as a gateway to implicitly flushing

the TLB as well. KAISER benefits from already existing implementations where TLB entries can be tagged with the CR3 register, which results in avoidance of frequent TLB flushes during context or mode switches. These implementations are presented by Venkatasubramanian et al. (2009). They explain this implementation is named Tag Manager Tables (TMT) and is used to optimize virtualization on x86 architecture systems. TMT is beneficial due to the low-latency characteristics. By utilizing TMT in the use-case of KAISER, the TLB is not as heavily affected in a negative way by the hardened isolation of user and kernel space.

3 Problem formulation

With the events of mid 2017 to 2018, the Meltdown vulnerability found in modern processors and the KAISER patch have raised many questions concerning mitigations and the effects thereof. This is understandable because the focus on virtualization is heavily changing and becoming more popular due to the methods of how people make use of computers today. Containers are changing the way of how virtualization is used between cloud providers and their customers. From the perspective of customers, a discovery such as Meltdown makes them question about the providers in terms of infrastructure design and information security/incident management. They want their information on the cloud to keep its confidentiality, integrity and availability. Where confidentiality is defined as a container not being available to anyone not authorized. Integrity is defined as the container contents not being leaked as a result of inefficient or jeopardized isolation. Availability is defined as its on-demand characteristics living up to a performance standard the customer is expecting and paying for. However, with every event there is a trade-off between benefits and drawbacks.

3.1 Research question

Gruss et al. (2017) tell that KAISER has the benefits of hardened isolation, which affects confidentiality and integrity positively, but has a drawback affecting availability in terms of performance. They do however justify that the maximum observed overhead impacting performance is merely 0,68% more than non-KAISER systems (PARSEC, pgbench and SPASH-2x are used as benchmarking tools). To counter this justification, the Meltdown whitepaper by Lipp et al. (2017) refer to KAISER as a temporary fix which contributes with a trade-off between performance and information security which has to be investigated in further research. Which raises the question for this final year project:

- What is the trade-off between performance and information security in container operations with KAISER enabled Linux kernels?

Although this work's topic is fairly unexplored at the time of writing due to the less matured areas of Meltdown and KAISER. It still means that as much as possible must be investigated around these changes in the world of virtualization. As of April 2018, Intel stated that several older microprocessors will not receive firmware updates, meaning that KAISER could potentially be relevant for a longer time than anticipated since hardware replacement is not always an option for everyone (Intel, 2018). Therefore this final year project aims to assist interested parties that are involved with lower-level computer usage-cases such as development, system administration, and computer engineering in Linux dominated environments. The intent is to present the potential existing overhead KAISER introduces by inspecting how it will affect the cost of system calls sent to the Linux kernel. In other words, this work has a point of view where containers are seen as a technology rather than a product in the context of KAISER and OS-level virtualization.

3.2 Related work

The fact that the area of this final year project is to this day not a mature subject makes it hard to compare with related work. However, the area of containers and virtualization is well documented and much research has been done with a huge amount of various topics. This final year project is firstly inspired by Morabito et al. (2015) who conducted a case study investigating the overhead differences between full and os-level virtualization in Linux based environments. They compared KVM, Docker, OpenVZ and LXC with native non-virtual computing instances and found out that KVM has a great performance with less overhead then expected, besides the disk I/O which can, assuming to them be a bottleneck. Besides this being the major overhead issues, they discuss the weight of containers is almost negligible. The source of inspiration was mainly due to the fact that the paper successfully mapped clear overhead differences between various virtualization tools. However, the happenings of Meltdown and Spectre stirred the performance numbers into an unknown direction. Ultimately, lowering the credibility of their results since their work was conducted during a time where Meltdown, Spectre and KAISER were not relevant. Simakov et al. (2018) inspects the KAISER patch effects on performance in "High Performance Computing" (HPC). This done by replacing the entire kernel with another one and conducting micro-benchmarks in single-node and multi-node setups. Their results describe performance losses depending on the type of workload, where single-node setups experienced 2-3% performance decreases, and multi-node setups experienced 5-11% worse performance. While this final year project does not aim to find results within the same area of topic, some commonalities can be taken into account. Simakov et al. (2018) used an experimental design which is applicable in reality, due to the fact that the entire kernel was replaced. Morabito et al. (2015) focuses on production-use products which are used in reality to build virtual environments. However the work of this final year project plans to skewer away the layers of excessive kernel functionalities and container features included in production-use technologies in order to minimize overhead.

3.3 Ethical aspects

If a performance loss is identified which originates from KAISER the interpretations could lead to performance being favored over information security. However, these results shall only represent a theoretical scenario and should never be seen as a justification of why such a use-case should be favored. If performance must be favored over information security, the administrator and other involved parties must exactly know what they are doing. This is because an insecure Linux kernel can jeopardize an entire host and its guests, which in turn can affect the whole business. The work of this final year project impartially intents to increase the transparency of what system security patches actually do to a system as a whole. This can be helpful to adaptation in for example application development, which is heavily reliant on performance of infrastructure in large-scale development projects, the same goes for network services in environments which must scale up to high availability. With such knowledge, older application code can be optimized and new code can be written in respect to information security and performance trade-off considerations introduced by KAISER.

4 Method

The method in this final year project follows the guidelines of conducting experiments described by Wohlin et al. (2012). Experiments in context of software engineering work well because it allows one to inspect the boundary between a phenomena within its context by manipulating and controlling variables in a systematical manner. Wohlin et al. (2012) explain that experiments can either be human-orientated or technology orientated.

The former is inapplicable in this final year project since the in boundaries and involved variables are all related to computer software components, making this experiment technology orientated. Case studies are well suited when a phenomena is difficult to distinguish from it's context and consist of an already existing set of variables which only need to be selected in order to create a "case". However, this final year project works with an experimental design which is better suited since all variables must be created, sampled and controlled specifically for the purpose of examining the cost of system calls generated from systemd-nspawn containers in pre- and post-KAISER environments.

4.1 Experimental design

By delimiting this final year project only to build-in functionalities of the Linux kernel, expected results should represent the fundamental core performance of a container in pre- and post- KAISER environments. The experimental design aims to meet this requirement by self-compiling a Linux Kernel rather than updating the host operating system entirely and using systemd-nspawn which lacks higher-level functionalities such as features explicit to Docker or LXC.

Data must be gathered, interpreted and presented. These requirements are met by first performing a baseline treatment, which is then followed up with a new treatment. This is further explained in chapter 4.1.1 and 4.1.2. The differences between the two treatments represent an overhead if the latter is larger than the former. This overhead is defined as a higher cost per system call. In the context of this final year project, the baseline describes the state of a computer system running Linux Debian 9 Stretch which has a KAISER disabled kernel. While the new treatment describes a state of the experiment in which one single variable is changed. In contrary, the baseline method consists of a KAISER enabled Linux kernel. This creates a context of a pre- and post-KAISER run-time environment in which benchmarks are conducted from within a systemd-nspawn container.

4.1.1 Variables

Independent variables are the tools and environmental characteristics which are controlled and manipulated in order to achieve an outcome also known as dependent variables. This experiment consists of two different independent variables. First the Linux kernel which changes its conditions and secondly a vast number of system calls will be recorded. The dependent variable is the time in nanoseconds (ns) of how long it takes to generate N number of system calls. System calls are an ideal measure point because they are used by a process to request privileged instructions. Dymshits, Myara & Tolpin (2017) explain that system calls are generated from all kind of activities performed on a computer. A modern computer handles hundred or more processes at the same time, where each process can produce thousands of system calls. If system calls were to be become more costly the overall impact on all these hundred or more processes would be negative. Ultimately meaning that the degradation of performance when KAISER is introduced depends on the number of system calls which are involved in a given type of workload. Their work introduces itself by stating system calls are a efficient representations of performance since it is a independent measuring point. No matter the type of activity the computer performs, system calls will always be involved one way or another.

4.1.2 Treatments

This experiment follows a experimental design which Wohlin et al (2012) call a ”one factor with two treatments”. Wohlin et al (2012) motivate that this design type is well fitting due to the simplicity and effectiveness of gathering and interpreting data by comparing the outcome of the treatments against each other. In this case, the independent variables are the Linux kernel and the system call benchmark tools, where the former changes throughout the experiment and the latter remains static. Independent variables can be static or be changed throughout the experiments, as long as their presence affects the dependent variables. Furthermore, Wohlin et al. (2012) explain; when changing an independent variable it becomes a factor. These factors have different values due to its changing nature which are in turn defined as treatments. By changing an independent variable such as the Linux kernel, it determines the treatments throughout the experiment. Meaning that two treatments are used throughout the experiment in order to determine the mean of the dependent variable. The treatments are; KAISER disabled and enabled Linux kernels. The dependent variable is the outcome of the treatment combined with static independent variables. In the context of this final year project' experimental design the dependent variable is the performance of a systemd-nspawn container which is defined as the the cost of a single system call in a pre- and post-KAISER environment (Wohlin et al., 2012).

4.2 Benchmarking

A benchmark for measuring performance in terms of execution time is necessary and must be well suited within the correct circumstances. These circumstances are met by using an appropriate type of benchmarking technique, which in this case must pressure the cost of system calls on a Linux-based computer system. This final year project will perform "micro-benchmarks" which will allow identification of the cost per system call. Na & Han (1999) tell that micro-benchmarks are ideal when measuring a single type of performance characteristic in computer systems. They continue to tell that such benchmarks are effective to identify, isolate and reproduce bottlenecks in computer systems. This is due to fact that few factors are involved in micro-benchmarks. In contrary, macro-benchmarks can be performed in order to identify performance on application levels since such benchmarks grasp the overall performance of a single application by including all involved factors. For example when measuring network performance, a micro-benchmark would only focus on latency or throughput. Whereas macro-benchmarks would look at the overall time it takes to execute a networking based application which involves both latency and throughput. This is inefficient when measuring the cost per system call because additional factors would deviate the dependent variable from the initial goal. The goal of this experiment is to identify the cost per system call between pre- and post-KAISER environments. Additional factors can make it difficult to identify the quantity of system calls and the cost of each individual system call if other dependent variables are present as well.

4.3 Systemd-nspawn container

As mentioned earlier in chapter 2.2, the Linux kernel comes with its own ability to create containers with systemd-nspawn. This means that systemd-nspawn’ core characteristics share many commonalities with the core characteristics of other technologies. This increases the relevance of the results presented by this final year project in the general context of OS-level virtualization.

The initial container from which the script is executed resides in a dedicated host. This host is subject to the treatments of the experiment in this final year project. The guest operating system is a AMD64 Debian Jessie image which is booted into a standard confined space which systemd has allocated during a bootstrapping process. This image is originally developed to run inside LXC containers, but due to the core commonalities between systemd-nspawn and LXC it is possible to boot the image within a confined space allocated by systemd (Linuxcontainers, 2018). All namespaces and cgroups are handled with default behaviour. Meaning that no additional configuration is done to change any of the namespaces or cgroup values for the container. In order to make sure the same container is used for each treatment the root directory is mounted on ”`~/kaiserContainerTree`” and the container is daemonized. This makes it possible to run the same particular container as a service on the host node no matter the treatments.

4.5 Used Tools

Because this final year project does not inspect any features explicit to certain Linux flavors, Debian 9 Stretch is used as the host’ operating system. The reason is because this is a widely available and well-known operating system with plenty of documentation. Because it is possible to change the treatments between reboots, only one node with a single hard disk is needed, ensuring that no hardware differences are introduced between the treatments. The total amount of hardware used during all treatments is listed in table 1.

HP Z400 Workstation
Intel Xeon W3550 @ 3.0GHz with HyperThreading enabled 24097MB Kingston DDR3 ECC Memory 1TB 5400RPM Western Digital Blue HDD

Table 1 - Hardware

Some additional software not included in a standard installation of Debian had to be installed specifically for the purpose of compiling a new kernel. The software that was added to the Debian 9 Stretch host is seen in table 2.

Name	Description
`build-essential`	Package used for building packages in Debian based operating systems
`linux-source`	The source code of the Linux kernel
`gcc`	Collection of compilers necessary for compiling a kernel
`libncurses5-dev`	Contains ncurses library which includes routines for the interactive selection screen used when creating a kernel config file
`libssl-dev`	Used for secure communication when communicating over the internet. Used when gathering packages from repositories
`bc`	Necessary for compiling the kernel
`lwm`	Windows manager used when using a interactive screen for kernel config creation

Table 2 – Tools installed for kernel compilation

In order to measure the exact amount of system calls a benchmarking tool developed by Tsuna found on Github is used (Tsuna, 2016). This tool proved to measure system calls in an effective way. This is compared to other tools such as ”strace” which can be found in the Debian repository. Strace is more fitting when identifying certain types of system calls a process generates, while the GitHub script proved to focus on system calls without categorizing them. This is more fitting for this final year project’ experimental design. The GitHub repository contains many different micro-benchmarking tools which cover context switch benchmarks but also system call benchmarks. The components consist of an application written in C combined with a Bash shell script which together generates 10’000’000 system calls and measures the summary time and the time per individual system call measured in nanoseconds (ns).

Each run will generate one set of total time and time per system call, which in itself does not prove much. This means that even if one result for each treatment can show a significant difference, it is not meaningful in a way that the numbers describe something practical or realistic. This is solved by generating a large set of results with mathematical characteristics allowing fabrication of a result representing a realistic scenario. Such generation is performed by creating a script written in Python3 that iterates one hundred times. Each iteration equals one single micro-benchmark which is written to a single .csv file which then is used for analysis. The number one hundred is chosen due to the time-consuming aspect of performing experiments. Since the variation between tests is not notable after a higher number than one hundred, a larger number did not seem necessary.

4.6 Threats to validity

Ensuring that the dependent variables are true and unbiased is an important aspect to consider when performing experimental studies. Therefore this chapter discusses some factors which potentially affects the validity of this final years project’ outcome.

The methodology of mitigating Meltdown in production environments is most likely defined by simply upgrading the operating system to a whole new kernel. However, in the case of this study, such a methodology threatens the internal validity since such kernel upgrades contain a wide variety of software packages not necessarily related to Meltdown or KAISER. This means that such a kernel potentially introduces superfluous factors which potentially change the outcome without knowledge thereof. In order to ensure the dependent variables are hold up scientific validity, the Linux kernel must be changed in such a way that no unknown factors arise. Therefore the default kernel included in Debian 9 Stretch is replaced with a self-compiled kernel with only the CONFIG_KAISER flag disabled or enabled in respect to the treatments. This toggle is supported and included in the Linux kernel source code since 2017 (Gruss, 2017).

Another preservation made for the internal validity was to make sure no unrelated services were running on the subject nodes. Therefore nothing was done over SSH but rather directly inside the bare-metal console of the physical node. Also, the running background processes such as ”cron” were identified with ”htop” and disabled if present.

The procedure to compile a new kernel with a custom set of features is not a cumbersome process due to the sophisticated set of tools involved. However, the presence of KAISER does not change the behavior of a computer system in a way that a user would notice right off the bat. Not obtaining this knowledge raises another validity threat. This time to the reliability (Wohlin et al., 2012). Therefore the kernel’s ability to provide a stronger isolation with KAISER is tested and confirmed by compiling a third party command line tool named ”Am-I-Affected-By-Meltdown” available on GitHub. This tool can identify how the kernel is mapped in relation to the user space on a system in order to detect vulnerabilities (Raphaelsc, 2018). This tool proved to be effective since the output showed ”System affected! Please consider upgrading your kernel to one that is patched with KPTI/KAISER” when the CONFIG_KAISER flag was disabled. And in contrary, the output showed "System not affected " each time the CONFIG_KAISER flag was enabled.

5 Results

This chapter focuses on the presentation of the achieved results from the initial experiment. First a pilot study is presented which justifies how the experimental design is appropriate and motivations to why certain changes were made. The initial experiment is then followed up. The results of the dependent variables for each treatment are presented in a graphical manner alongside with an explaining text and table.

5.1 Pilot experiment

On the first try of performing benchmarks, a macro-benchmark tool was used to identify if the earlier statements about micro-benchmarking hold up in reality. This involved a Python3 script that generated many different types of system calls all in relation to network performance, disk performance, CPU and memory utilization. The results of this script generated a lot of different system call with a high standard deviation originating from 200 executions. After many tries of interpreting these results, it proved to be difficult to locate the origin of such behavior. Although the unfortunate results, the statement about macro-benchmarks in chapter 4.2 holds up in reality which is an indication that the overall experimental design presented in chapter 4.1 can be used for the main experiment.

5.2 Main experiment

The first inspected treatment is a KAISER disabled Linux kernel. The results showed different numbers compared to the pilot experiments due to different benchmarks. The presentation here represents the results from the treatments in the form of a graph respective to the cost per system call in a sequence of 10’000’000 system calls which occurred during one hundred iterations. These graphs are complemented with tables containing maximum and minimum observed numbers alongside with standard deviations and average numbers.

5.2.1 Main experiment - Results

Both the treatments are seen in figure 6 and table 3. The overall time for both treatments was not persistent throughout each iteration. However, this is not a decisive characteristic since the importance of these results lies in the differences between the two treatments. It can be observed that a large difference exists between the two. The average time per system call of the baseline is ~814ns whereas the new treatment indicates an average of ~1093ns. The standard deviations of the baseline and the new treatments are 2,032ns and 7,1ns respectively. What these results mean in the context of this experiment is that a one hundred iterations of a sequence of 10’000’000 system calls between both treatments indicated that an average difference of 29,3291% per system call is introduced by implementing KAISER.

These differences do indicate that each individual system call became more costly in terms of needed time when the new KAISER enabled Linux kernel was implemented. In terms of a trade-off between information security and performance, this means one thing. If a computer runs many system call intensive tasks, the trade-off would likely impact the performance in a negative way. The total impact depends on the number of system calls. Which consequentially could impact the purpose of the computer system negatively overall depending on what the purpose actually is. For instance, web servers which serve websites to thousands of users simultaneously generate a lot of system calls. Such a scenario could potentially lead to the end-users or customers experiencing an increase in response time if the web server has a KAISER enabled Linux kernel. However, if the contrary were true, end users would likely not experience a severe performance impact at all.

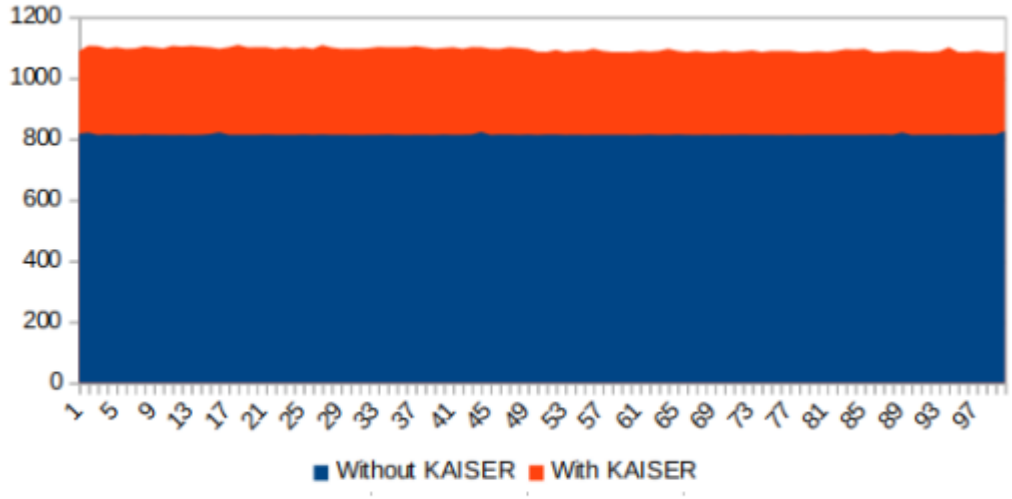

Figure 6 – Main experiment results (X = Iterations, Y = ns)

Time = ns	**KAISER disabled**	**KAISER enabled**
Avg.	813,931	1093,672
Std.	2,0332498338	7,101103627
Max.	825,5	1108,9
Min.	812,9	1082,7

Table 3 – Main experiment summary results

6 Conclusion

Essentially each time a container wants to utilize kernel functionalities it has to wait. The waiting time is introduced by system calls because they add an overhead since the container has to ask the kernel for help. The waiting time per system call proved to increase with around 29% when the Linux kernel was secured by removing kernel space mappings from user space which is possible with the KAISER patch. This fact answers the research question of this final year project. Earlier in chapter 3 the terms confidentiality, integrity, and availability were mentioned in relation to end-user expectations where the last one raised curiosity around the investigated research question. However, the results represent time measured in nanoseconds per system call. This seems like such a small fraction of what an entire container is using when active. Although when considering for example; on average a certain container utilizes around 9'000 system calls for a single set of instructions where each system call costs ~1093ns for the application it is housing. The performance impact which represents the availability an end user is expecting sums up to ~9 837 000ns or ~0.009837 seconds. In itself these numbers do not seem like much, but when serving information to end user this is sufficient enough to change the experience of the service if hundreds or thousand concurrent sources want connect to one single container. This would be unpleasant if for example the application is included in the backend of time-intensive system such as economical stock monitoring software.

Simakov et al. (2018) performed tests across several different computer nodes because their research area involved clustered environments. They came to the conclusions that the performance across one single node decreased around 2-3%. While the performance across several nodes decreased by 5-11%. Their treatments were similar to this final year project' treatments since they inspected the patch effects related to Meltdown. However, they never explicitly tell that KAISER is indeed the patch in question. The major difference between Simakov et al. (2018) experimental process and the experimental process of this work are the tools used. Simakov et al. (2018) used benchmarks which share the characteristics of macro-benchmarks. Chapter 4.2 explains that such benchmarks are ideal for investigating the overall performance of applications. The 29% performance difference observed in this final year project seems reasonable when compared to Simakov et al. (2018). Because of what Dymshits, Myare & Tolpin (2017) say about system calls; how system calls are generated from every activity performed on a computer. This potentially means that the origin of their observed performance loss is due to the performance difference observed in the cost per system call seen in this work.

7 Discussion

This chapter discusses the results followed by general thoughts about the experimental process, limitations, and reflections. This study aims at interested parties that are involved with lower-level computer usage such as development, system administration, and computer engineering in Linux environments. The observed performance difference of ~29% represents the results of this final year project. This indicates that a performance loss exists between a KAISER disabled an enabled Linux kernel if information security is favored over performance. These number successfully answer the research question. Also due to the choice of performing simple and low-level experiments this work even partially managed to answer where the cause of performance degradation in the KAISER patch originates from. Why partially is because system calls do not venture all the characteristics of the KAISER patch, this could, for example, be complemented by investigating context switches.

Even though the performance difference indicates towards a negative impact, these results should not be seen as a justification to prioritize performance over information security. The people administrating and owning systems who would find themselves in a situation where such a choice is an obstacle should carefully identify why performance should be favored over information security. This can be done by analyzing the purpose of the system and the affected assets by performing a risk analysis. This is because an insecure Linux kernel allows the applications running inside containers to break the isolation and in turn result in leakage of information if KAISER is not implemented. This has been observed by Lipp et al. (2017) during their Meltdown research. Such events can jeopardize the whole operation of a company, consequentially affecting a larger group of people such as communities, country regions or whole countries depending on what type of information is at stake. Although the disclaimer, these results could in turn be used as a motivation to optimize existing systems and developing more efficient modern systems which can meet both performance and information security. By using the 29% performance difference of system calls with a KAISER enabled Linux kernel, software developers can create optimized software which reduces the amount of involved system calls per activity. Also system administrators and information security experts can perform risk analysis with better understanding of what a non-patched system means in terms of information security and performance. Those goals contribute to both the modern society which heavily relies on digitalization and the modern IT industry in general.

7.1 Experimental process

The experimental process of final year project used micro-benchmarking in the context of a technology orientated experimental study. This choice of benchmarking is optimal when a small characteristic of a n application must be investigated. This is a statement that proved itself to be correct at the time of conducting the experimental process. This is because first a pilot experiment was performed which consisted of the contrary. The outcome thereof showed inappropriate results which made it a cumbersome task to interpret them in a sensible way. While an experimental study is an obvious choice when a quantitative result is expected, the choice between performing case studies or experimental studies is not obvious. This requires a careful analysis of what outcome is expected and how such a result is reached. Due to the fact that the cost of system calls measured in time is a specific type of outcome, the variables have to be well adapted to such an outcome. Wohlin et al. (2012) tell that the variables used to create an outcome are called independent variables, which in turn can change during the experimental process but don't necessarily have to. However, if independent variables change they are defined as factors, where each value after or before a change is seen as a treatment. Meaning that this final year project' method is a quantitative experimental study consisting of one factor with

two treatments. Which proved to be an effective method due to the concrete results clearly showing that each treatment affected the cost per system call. During the design of the experimental process, it became clear that all involved variables and tools should be as closely related to the core functionalities of the Linux kernel in order to avoid superfluous factors. In the end, the only tool that was not closely related to the Linux kernel was the script used to generate 10’000’000 system calls. Although, the script did generate system calls which are directly sent to the kernel and is written in the low-level programming language C this is not a concern. This script was compared with a tool named ”strace” which is included in the Debian Stretch repository. However, this tool proved to only introduce extra complications since it focused more on the categorization of system calls and less around the generation of an exact amount of system calls.

The operating systems involved in the experiment were all Debian based. Where the host ran Debian Stretch and the guest ran Debian Jessie. The main difference between the two is that the latter is an older version than the former. Debian was chosen since it is a widely available Linux distribution with a large amount of documentation. This allowed a fall-back point if something were to go wrong during the treatments, since the replacement of a Linux kernel could lead to some serious technical implications.

7.2 Limitations

Meltdown is the main focus of this final year project due to two reasons; 1) KAISER mainly only mitigates Meltdown due to the characteristics of the vulnerabilities. 2) Spectre is a more complex issue which is not as easily mitigated with KAISER. This is important to note since both Meltdown and Spectre are often talked about within the same context, which is not the case with this work. The results presented in this work are not applicable within the context of Spectre. Regardless of those limitations, this final year project managed to answer the research question which in turn meet the aim of mapping transparency for the trade-off between information security and performance when KAISER is implemented. By knowing that system calls have become more costly in terms of time, applications can be optimized to utilize system calls more sparingly.

7.3 Reflections

In hindsight, there is no evidence if the container actually became less lightweight. However, there are no regrets of including the aspects of containers and virtualization as a whole. This has benefits to this work since this contributed to achieving a broader scope in which both Meltdown and KAISER are relevant. Similar implementations on different systems could lead to the same results in terms of percentual difference. Such systems could describe bare-metal (non-virtualized) or traditional virtual machines. With that being said, the results presented here originate from an experiment tailored specifically for this final year project. This means that the numbers are only a representation of performance per system call in the context of the experimental process used in this final year project. Also, the results should not be interpreted as the overall performance of a computer system but rather that system calls are a changing factor when considering performance over information security when implementing a Linux kernel.

8 Future work

Further research could investigate how context switches behave after KAISER implementations. These are mechanics that are close to both user space and kernel space, which as mentioned earlier in this final year project change quite drastically when KAISER is enabled inside the Linux kernel. Such as study would complement this final year project. Since the results shown here slightly indicate where the performance degradation originates from, however, such a statement would be biased if context switches are not fully investigated as well.

Another experimental study which is more in-depth than this work would be possible by conducting more specific micro-benchmarks. By taking a look at specific types of system calls related to the network I/O, disk I/O or file handling etc. with the aim of identifying the most expensive type of system call.

As mentioned earlier in the conclusions. The performance impact on the whole system and the end user's experience all depends on the number of system calls generated per sequence of instructions. Therefore it is worth mentioning that future work which aims to investigate a similar performance trade-off for a single application could benefit from the findings of this final year project. Such an application could, for example, be a database.

At last, it should be mentioned that performance with respect to Spectre should be investigated as well. Such as study should be an experimental study which can borrow many aspects of the method used in this work. The differentiating characteristics of such a study should in such case be another set of variables due to the different nature of how Spectre's mitigations work.

References

Dymshits, M. Myara, B. Tolpin, D. (2017). Process monitoring on sequences of system call count vectors. in Ortega-Garcia, J. (eds.) on 2017 International Carnahan Conference on Security Technology (ICCST), Madrid, Spain, 23-26 Oct. 2017 , pp. 1-5, IEEE, DOI: 10.1109/CCST.2017.8167792

Edge, J. (2013). Creating containers with systemd-nspawn, https://lwn.net/Articles/572957/ [02-05-2018]

Grattafiori, A. (2016). Understanding and Hardening Linux Containers. [Blog] 20 April. https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/april/understanding-and-hardeninglinux-containers/ [2018-02-21]

Gruss, D. (2017). KAISER - do not map kernel in user mode. https://lkml.org/lkml/2017/5/4/220. [208-03-28]

Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., Mangard, S. (2017). KASLR is Dead: Long Live KASLR. In Bodden, E. Payer, M. & Athanasopoulos (eds.) on 9th International Symposium on Engineering Secure Software and Systems, ESSoS 2017, 3-5 July 2017. Bonn, Germany, pp. 161-176, Springer-Verlag Italia, DOI: 10.1007/978-3-319-62105-0_11

Gruss, D. Maurice, C. Fogh, A.(2016). Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In Weippl et al. (eds.) on 23rd ACM Conference on Computer and Communications Security (ACM CCS 2016), Vienna, Austria, 24-28 October 2016. Vienna, Austria, pp. 368-379, ACM New York, DOI: 10.1145/2976749.2978356

Hund, R. Willems, C. Holz, T. (2013). Practical Timing Side Channel Attacks against Kernel Space ASLR. In Kellenberger, P. (eds.) on 2013 IEEE Symposium on Security and Privacy. Berkeley, CA, USA, 19-22 May 2013. pp. 191-205, IEEE, DOI: 10.1109/SP.2013.23

Intel. (2018). microcode revision guidance. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/04/microcode-update-guidance.pdf. [2018-04-13]

Jang, Y. Lee, S. Kim, T. (2016). Breaking Kernel Address Space Layout Randomization with Intel TSX Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In Weippl et al. (eds.) on CCS '16 Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna, Austria, 24-28 October 2016. Vienna, Austria, ACM New York. 380-392. DOI: 10.1145/2976749.2978321

Kernel.org. (2017). Index of /pub/linux/kernel/v2.6/ https://mirrors.edge.kernel.org/pub/linux/kernel/v2.6/ [2018-0502]

Kerr, L. & Alves-Foss, J. (2016). Combining Mandatory and Attribute-Based Access Control, in Ung, X. B. & Sprague, R. H. Jr (eds.) on 2016 49th Hawaii International Conference on System Sciences (HICSS), Koloa, HI, USA, 5-6 January 2016, pp. 2616 – 2623, IEEE, DOI: 10.1109/HICSS.2016.328

Kerrisk, M. (2013). Namespaces in operation, part 1: namespaces overview. https://lwn.net/Articles/531114/ [2018-02-27]

Linuxcontainers.org. (2018). Image server for LXC and LXD. [ONLINE] Available at: https://us.images.linuxcontainers.org/. [2018-02-09]

Lipp, M. Schwarz, M. Gruss, D. Prescher, T. Haas, W. Mangard, S. Kocher, P. Genkin, D. Yarom, Y. Hamburg, M. (2018) Meltdown. https://arxiv.org/pdf/1801.01207v1.pdf [2018-05-31]

Marmol, V. Jnagal, R. Hockin, T (2015), Networking in Containers and Container Clusters', in The Technical Conference on Linux Networking, Ottawa, Canada, February 14-17, pp 1-4, http://wiki.iptables.org/pablo/netdev0.1/papers/Networking-in-Containers-and-Container-Clusters.pdf

Microsoft. (2018). Interprocess Communications. https://msdn.microsoft.com/en-us/library/windows/desktop/aa365574(v=vs.85).aspx [2018-05-01]

Morabito, R. Kjällman, J. & Komu, M (2015). Hypervisors vs. Lightweight Virtualization: A Performance Comparison. In Calton, P. & Ajay, M. (eds.) on 2015 IEEE International Conference on Cloud Engineering, Tempe, AZ USA, 9-13 March 2015, pp. 386-392, IEEE. DOI: 10.1109/IC2E.2015.74

Na, S. & Han, D. (1999). Microbenchmarks of operating system performance on the multiprocessor platform. on TENCON 99. Proceedings of the IEEE Region 10 Conference 5-17, Cheju Island, South Korea, Sept. 1999, pp. 435 – 438, IEEE, DOI: 10.1109/TENCON.1999.818444

Raj MP, A. Kumar, A. J Pai, S. Gopal, A, (2016). Enhancing security of Docker using Linux hardening techniques. In Niranjan, S.K. (eds.) on Applied and Theoretical Computing and Communication Technology (iCATccT), 2016 2nd International Conference on. Bangalore, India, 21-23 July 2016, pp. 94-99, IEEE. DOI: 10.1109/ICATCCT.2016.7911971

Redhat. (2018). 4.2. SELinux and Mandatory Access Control (MAC). https://access.redhat.com/documentation/ en-us/red_hat_enterprise_linux/7/html/virtualization_security_guide/ sect-virtualization_security_guide-svirt-mac. [2018-05-09]

Redhat. (2018). 1.2. Relationships Between Subsystems, Hierarchies, Control Groups and Tasks. https://access.redhat.com/documentation/en us/red_hat_enterprise_linux/6/html/resource_management_guide/secrelationships_between_subsystems_hierarchies_control_groups_and_tasks. [2018-05-05]

Scheepers, M. J. (2014) Virtualization and Containerization of Application Infrastructure: A Comparison. in 21st Twente Student Conference on IT June 23, Twente, The Netherlands, 23 June 2014, pp. 1-7, The University of Twente

Simakov, N., Innus, M.D., Jones, M.D., White, J.P., Gallo, S.M., DeLeon, R.L., & Furlani, T.R. (2018). Effect of Meltdown and Spectre Patches on the Performance of HPC Applications [Preprint]. Available at: https://arxiv.org/abs/1801.04329v2 [2018-05-24]

The Linux Foundation (2013) [video]. Systemd-Nspawn is Chroot on Steroids - Lennart Poettering https://www.youtube.com/watch?v=s7LlUs5D9p4 [2018-05-02]

Turnbull, J. (2006). DAC and MAC safety. http://searchdatacenter.techtarget.com/answer/DAC-and-MACsafety. [2018-05-09]

Venkatasubramanian, G. Figueiredo, R.J. Illikkal, R (2009) TMT - A TLB Tag Management Framework for Virtualized Platforms. On 2009 21st International Symposium on Computer Architecture and High Performance Computing, Sao Paulo, Brazil, 28-31 Oct. 2009, pp. 153-159 IEEE, DOI: 10.1109/SBAC-PAD.2009.21

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A. (2012). Experimentation in Software Engineering. 1st ed., Berlin: Springer Heidelberg New York Dordrecht London. DOI:10.1007/978-3-642-29044-2

Wright, C. Cowan, C. Morris, J. Smalley, S & Kroah-Hartman, G, (2002). Linux Security Module Framework. In Boneh, D. (eds.) on Proceedings of the 11th USENIX Security Symposium, CA, USA, August 05 - 09, 2002, pp. 17-31, USENIX Association Berkeley

Raphaelsc. (2018). Am-I-affected-by-Meltdown. https://github.com/raphaelsc/Am-I-affected-by-Meltdown. [2018-04-04]

Tsuna (2016). tsuna/contextswitch. https://github.com/tsuna/contextswitch [2018-05-09]

Appendix A – systemd-nspawn configuration

Bootstrapping

```
sudo bootstrap –arch=amd64 jessie ~/kaiserContainerTree
```

Starting container (-b stands for booting the guest OS, only works if guest has support for systemd)

```
sudo systemd-nspawn -D ~/kaiserContainerTree test_container -b
```

daemon/service file contents inside host path: `/lib/systemd/system/kaisercontainer.service`

```
[Unit]
Description=Test Container

[Service]
LimitNOFILE=100000 # limit to max. open files by container
ExecStart=/usr/bin/systemd-nspawn --machine=test_container --directory=/
var/lib/machines/container1/ -b
Restart=always # restart container after a crash

[Install]
Also=dbus.service # necessary to open PTY (used when entering container
TTY)
```

Additional software packages installed inside container:

`dbus, python3, python3-pip, sudo`

Appendix B – Scripts

https://github.com/tsuna/contextswitch

Systemcall micro-benchmarking tool

```
// Copyright (C) 2010  Benoit Sigoure
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.

#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <time.h>
#include <unistd.h>

static inline long long unsigned time_ns(struct timespec* const ts) {
  if (clock_gettime(CLOCK_REALTIME, ts)) {
    exit(1);
  }
  return ((long long unsigned) ts->tv_sec) * 1000000000LLU
    + (long long unsigned) ts->tv_nsec;
}

int main(void) {
  const int iterations = 10000000;
  struct timespec ts;
  const long long unsigned start_ns = time_ns(&ts);
  for (int i = 0; i < iterations; i++) {
    if (syscall(SYS_gettid) <= 1) {
      exit(2);
    }
  }
  const long long unsigned delta = time_ns(&ts) - start_ns;
  printf("%i system calls in %lluns (%.1fns/syscall)\n",
         iterations, delta, (delta / (float) iterations));
  return 0;
}
```

run.py – Used to automate all 100 iterations

```
#!/usr/bin/python3
import re
import subprocess
import csv
import sys
from time import sleep

re_bench = re.compile(r'in\s([0-9]+)ns\s\(([0-9.]+)ns/syscall\)')

def fetch():
    # run the experiment
    # by spawning a workload script as a seprate process
    data = subprocess.Popen(
        ['./timesyscall'],
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE
    )
    out, err = data.communicate()
    return out, err

def data_handler(syscall_perf):
    # manipulate the outcome and write to csv

    # Samples:
    # ('1708883448', '170.9'), ('1737738060', '173.8')]
    context_fp = './syscall_perf.csv'

    s_file = open(context_fp, 'w')
    syscall_perf_file = csv.writer(s_file)
    syscall_perf_file.writerow(['Run', 'total_ns', 'ns/syscall'])

    count = 1 # context run count
    while True:
        for i in syscall_perf: # Write context switches information to csv
            syscall_perf_file.writerow([count, i[0][0:11], i[1]])
            count += 1
        break

def main():
    perf_total = []
    for i in range(100): # number of times the experiment runs
        sleep(1)
        syscalls = fetch()[0]
        sysperf = re.findall(re_bench, syscalls.decode('utf8'))
        perf_total.append(sysperf[0])
    data_handler(perf_total)
    sys.exit()

main()
```

Appendix B – Compiling a Linux kernel

By following the following steps in order it is possible to compile a kernel in a similar manner as was done in this final year project.

1. Installing the kernel source code

```
~$ sudo apt-get update
~$ sudo apt-get install linux-source
~$ cd /usr/src/
~$ sudo tar xjvf linux-source-<version>.tar.bz2
~$ cd linux-source-<version>
```

2. Installing Prerequisites

```
~$ sudo apt-get install build-essential gcc libncurses5-dev
libssl-dev
```

3. Creating a configuration file

```
~$ sudo make nconfig
```

Navigate to the security menu and select or remove the parameter seen in Figure 7, then save by pressing F9.

4. Compile and install the kernel

```
~$ sudo make -j$(nproc –all) # This utilizes all cores

~$ sudo make modules_install

~$ sudo make install
```

Reboot

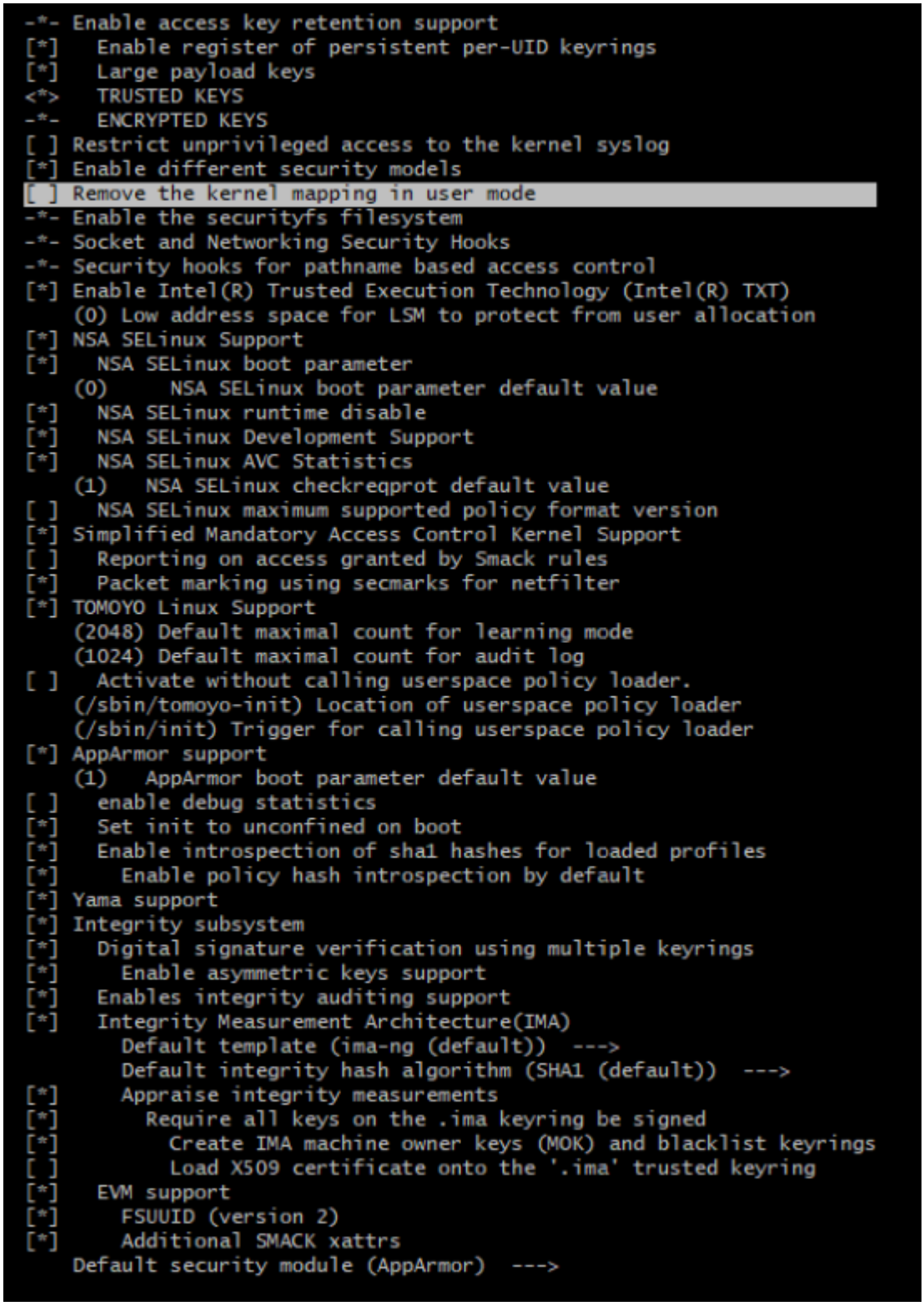

Figure 7 - nconfig