Spectre and Cloud : An evaluation of threats in shared computation environments

Mundhenke, Moritz
Mannheim University of Applied Sciences
Department of Computer Science
Paul-Wittsack-Str. 10, 68163 Mannheim

Abstract—**The processor flaws used in the Spectre and Meltdown attacks have had uncharacteristically large media impact, even gaining coverage in main-stream media. This is despite the fact that this type of exploit has not been used in any real world attacks and is unlikely to target consumers, as simpler attack vectors still remain highly effective. However, because Spectre affects any processor which uses speculative execution, with little hope for a "silver bullet" in the near future, Spectre seems to be here to stay.**

While Spectre might not be very relevant to the consumer market, it is quite relevant where safety is usually paramount: the cloud.

It promises cost reduction and safety through offloading maintenance and updating tasks to gigantic providers like Amazon's AWS. But how secure can the most up-to-date platform be, if the used hardware is inherently flawed to the core?

This paper provides a high level explanation of the Spectre attack, shows potential Spectre attack vectors in a shared cloud environment and discusses some defensive measures.

Contents

1. Introduction

Cloud computing can offer enormous cost savings by reducing administrative effort for the customer and making use of hardware more efficiently by maximizing utilization. To achieve this increased utilization, hardware has to be shared in some form between multiple customers [30]. In combination with the hardware flaws uncovered by the Spectre attacks and the potential for anyone to buy cloud resources, this creates an entirely new threat model for cloud applications.

To investigate this problem we first discuss the technologies used in today's computation environment that play a part in Spectre attacks. Afterwards compute cloud offerings are differentiated and potential attack vectors shown. This is done through a combination of reviewing past attacks on cloud infrastructure and potential application of Spectre exploits in these scenarios to determine the associated risks. These risks remain entirely theoretical as experimental proof of such attacks is outside of the scope of this work.

Finally mitigation options are discussed and some recommendations based on the findings of this work are given.

2. Modern computation hardware and infrastructure

2.1. Memory management

Since around 1980 memory performance has not been able to keep up with the ever increasing processor speeds. [23, p. 73] Because of this accessing the main memory has become a very time consuming task. Modern processors therefore try to circumvent RAM access through the use of multiple cache layers (see Figure 1). These caches have a rather limited capacity especially the very fast L1 cache (Skylake i7-6700 L1 cache 2x32KB per core, L3 cache 8MB total) [3].

To optimize the usage of the available caches, various techniques are used. When code accesses a memory address, such as an array or string, it is very likely that addresses nearby will also be accessed (*spatial locality*). Because of this, memory is loaded into the caches in blocks to reduce cache misses in these cases. [23, p. 74].

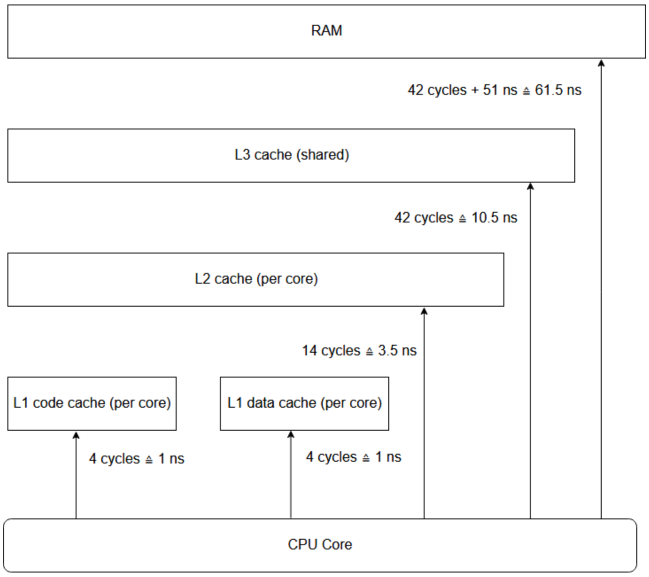

Figure 1. The Intel Skylake cache hierarchy and access latencies [21]. Timings are calculated assuming a clock speed of 4 GHz for an Intel i7-6700 processor using DDR4-2400 RAM [3].

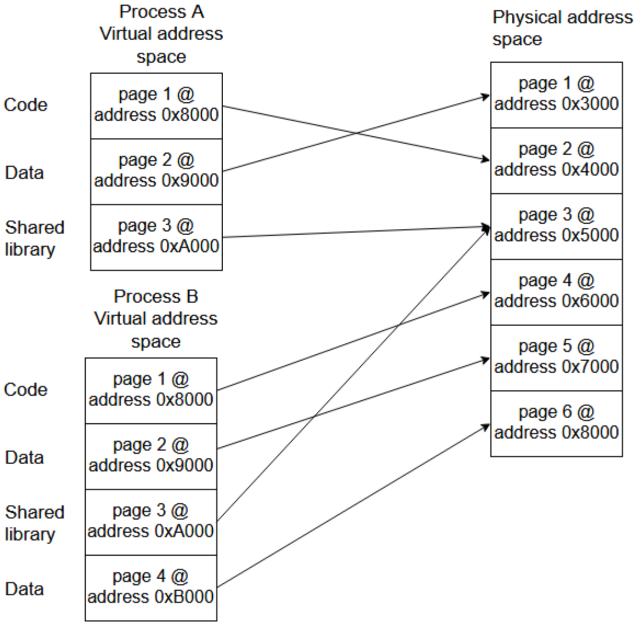

Figure 2. Mapping of virtual memory to physical memory [37].

2.2. Processor pipelines and speculative execution

Since the turn of the millennium, clock speeds of high performance CPUs have stagnated because the increased energy consumption and resulting heat generation got too high [21]. Multi-core architectures became common, however even today many, if not most, workloads rely heavily on single thread performance. Therefore, to increase the speed of single thread execution, without increasing the clock speed, more instructions have to be executed in a single cycle. To achieve this goal many techniques like instruction splitting, fusion, and simultaneous instruction execution are used [21]. These optimizations require more complex processing of the instruction stream which is handled by the processors instruction pipeline. This has resulted in higher pipeline memory requirements as well as a longer pipeline in terms of clock cycles required for an instruction to complete execution. However, since programs use conditional branches and loops these long pipeline can become invalid because the CPU cannot know the destination of a conditional jump without first evaluating the condition. This results in a pipeline flush that wastes valuable execution time while the pipeline has to be refilled. To mitigate the impact of this issue the processor can store previous outcomes of branches and use this to predict future executions of the branch. With branch prediction only a miss-prediction will result in a pipeline flush. The CPU runs the instruction of the predicted branch and either commits the results if the prediction was correct or discards them if it was not. This process is called speculative execution [28].

2.3. Virtual and shared memory

Running multiple programs in parallel is a operating system "feature" that has been the standard for around 30 years. Before that some operating systems would run only one program at a time or completely swap out the main memory when switching to a different running process [37]. This method is obviously not very efficient at handling many processes at once. Today, virtual memory is the primary way of managing memory. When using virtual memory every process has it own separate address space. This means two processes can store different data at the same virtual address without being able to affect each other. [23, p. 620]

The virtual address space of each process is divided into memory blocks called pages (currently on x86 computers page size is 4kB). When a process accesses an address, the virtual page address is translated by the memory management unit (MMU) to a physical frame address or, when the page is not loaded into memory, gets loaded into memory by the OS.

While virtual memory allows reuse of the same virtual address by multiple processes, it also allows sharing of physical memory between processes. This can be used for inter process communication and also as an optimization to share identical code between programs. Shared libraries are loaded once and can be reused by any program that uses the same library version [37]. To prevent programs from changing the library code for other programs, a copy on write mechanism is used. When a process attempts to write to a shared library page, it is copied in physical memory and the virtual address gets remapped to the copied version. [12, p. 295]

2.4. Docker

Containerization concept. Docker allows the isolation of programs in containers. These containers hide other processes running on the host and provide the process with its own virtual filesystem and network. This means containers only share the hosts kernel which stands in contrast with classical VMs which have to run a separate OS for each virtual machine [10].

To create a container an image is used. The image contains all required files and serves as a "blueprint" for the container [10]. Due to the lightweight nature of containers it is possible and best practice to isolate every application in its own container [11]. For example a classic PHP and MySQL application, also called LAMP stack (Linux, Apache, MySQL and PHP), would use a MySQL and an Apache container. This has multiple benefits: Developers can build and test container images on their own machine which will behave identical when run on a production server. Docker also allows easy horizontal scaling of applications. The LAMP stack mentioned above could be extended by creating an additional Apache container and an nginx container to load balance between the two Apache servers.

File-system layers. To reduce redundancy docker images are divided into layers [11]. For example an Apache PHP application would consist of 3 major layers. The first contains a minimal debian image providing basic libraries and tools. The second layer contains Apache and its dependencies. The final layer would then have a copy of the applications PHP source files.

When a container is started from this image, the layers are stacked on top of each other using a union filesystem like OverlayFS [33]. Whenever the application attempts to read a file, the kernel checks each layer from top to bottom returning the first result. This allows overriding of files on lower layers.

Because of this multiple different PHP images can share the first two layers and a MySQL image sharing the base Debian image. Docker also adds a final non-persistent layer to the container which the application running inside uses to write files, which is discarded when the container is shut down.

2.5. Cloud

National Institute of Standards and Technology (NIST) defines cloud computing as "[...] a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [30]. Further more 5 essential characteristics are defined: On-demand self-service, broad network access, resource pooling, rapid elasticity and measured service. Resource pooling is of special interest in the context of this work.

Cloud resources are shared by multiple costumers in a multi-tenant model [30]. Concretely this means that tenants (customers) of a cloud platform share a server provided by the platform. This might be in form of quite direct sharing of a server by running virtual machines owned by different tenants on it. However, more abstract offerings like Software as a Service (SaaS) databases, which give the customer no direct control over an operating system, will have some form of resource sharing. This results in customers having to trust not only the confidentiality of the provider itself, but also trust the isolation that separates them from potentially malicious third parties.

3. Spectre

3.1. Side-channels

Computer programs can, on an abstract level, be described with the input-process-output (IPO) model. An input will produce an output defined by the algorithms of the program. This output is deterministic, meaning input `a` will always result in output `b`.

However in real life systems, processes can produce additional outputs which are not intended by its programmer. For example a program which calculates the factorial of a number takes longer proportional to the input number. An attacker could therefore, infer the input number from the execution time without knowing the result. More complex timing-attacks have been used to extract private keys or other secrets of various cryptographic algorithms [9][26]. Similarly even a system's power consumption can be used as a side-channel to retrieve secret data [27].

While these two side-channels can add unintended output to a process they are within the programmers control. Execution time can be increased arbitrarily through idling or sleeping and the system's power consumption can be increased by running pointless code. Other side-channels however, are completely outside of the control of the programmer. For example a program has very limited or no direct control over the systems cache state or which arithmetic logic unit (ALU) will compute a calculation [28].

3.2. Exploiting speculative execution

To execute a Spectre exploit the attacker first has to find a vulnerable instruction sequence in the victim program. The sequence will leak secret data into the chosen side-channel during speculative execution. Therefore, Spectre-style attacks can be differentiated from each other by the how speculative execution is achieved and what side-channel is used [28].

```
if (x < array1_size)
  y = array2[array1[x] * 4096];
```

Listing 1. Spectre Variant 1 example [28]

Variant 1 of the original Spectre attack uses conditional branches to achieve speculative execution and leaks data through the processors cache state. The conditional branch is an array bounds check `if (x < array1_size)`. The branch prediction can be miss-trained by repeatedly calling the victim function with an in-bounds `x`, causing it to predict the branch as true in the future. Then the attacker flushes `array1_size` from the processors cache and calls the victim function again. This causes the branch to be execute speculatively until `array1_size` is retrieved from memory. To leak the victims secret the attacker chooses an `x` that is out of bounds. The victim now reads from `array1` using the malicious, out-of-bounds address. To successfully execute the attack sequence the victim also has to use the result to access a second array. This causes the processor to load a memory address into the cache which is based of the previous out-of-bounds access of `array[x]`.

While the direct results of the speculative execution are discarded, the attacker can now measure their own access time of the possible memory addresses of `array2`, revealing which address can be read fastest and was therefore retrieved by the victims speculative execution, revealing the (secret) value of `array[x]`.

4. Spectre in shared cloud infrastructure

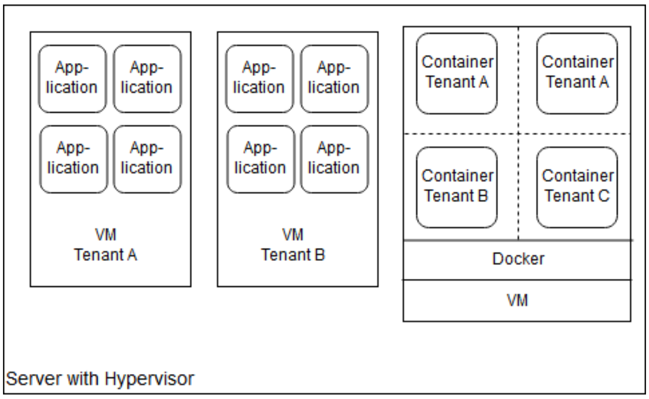

Figure 3. Tenant isolation in a cloud environment using a hypervisor and using Docker container isolation.

4.1. Shared hypervisor

Infrastructure as a Service (IaaS) providers give their customers, here referred to as tenants, the ability to rent virtual machines. The tenant has full access to this VM (root access). Since most applications do not require the full capabilities of the physical server, multiple VMs are run on a hypervisor such as VMware ESXi or Linux's KVM.

This results in the sharing of physical resources like RAM and CPU cores between tenants. This isolation is hardware enforced and hypervisor exploits are quite rare (only 5 ESXi vulnerabilities in 2018 with a Common Vulnerability Scoring System (CVSS) score ≥ 5) [38].

However, since speculative execution can circumvent hardware enforced checks, some Spectre vulnerabilities allowed an attacker to read the memory of other tenants virtual machines and the hypervisor itself [40, 39].

While the currently known Spectre attacks have been mitigated by hypervisor patches, speculative execution exploits found in the future could still allow cross VM attacks in the cloud.

In a public cloud an adversary is faced with another hurdle besides VM isolation. Because the IaaS provider allocates resources for the tenants based on the their requirements through a closed-source algorithm, an attacker can not directly force co-residency (share a hypervisor) with the victim.

Ristenpart et al. [35] demonstrated various networking based techniques to achieve co-residency in the early (2009) Amazon Web Services (AWS) cloud and while Amazon has improved security in this regard, other co-residency detection mechanisms have been found [25]. This newer method uses information leaks on the processor level, specifically through the last level cache shared between all CPU cores.

4.2. Shared Docker host

Running docker containers in the cloud can have benefits for both the customer and the cloud provider. The customer does not have to worry about maintenance of the underlying operating system and can more easily scale their application. The cloud provider can, in theory, increase utilization of their hardware because of the reduced overhead.

However, leveraging these benefits requires faith in the process isolation provided by Docker and the Linux kernel. Because of this the Azure and AWS public cloud are using hypervisor technology to isolate containers, only sharing kernels between containers of the same user defined application group [4, 8]. Due to this the potential price advantage is lost. The smallest possible AWS Fargate task configuration (0.25 vCPU and 0.5 GB RAM) costs 0.01234 USD per hour [5] while the EC2 t3.micro VM (2 vCPU and 1 GB RAM) only costs 0.0104 USD per hour [19].

Other Platform as a Service (PaaS) providers like Heroku [24], OpenShift [15] and the defunct DotCloud however, do not use the additional isolation provided by hypervisor virtualization [42].

These providers do refer to various "Docker hardening" techniques, however, specifics like the usage of shared libraries between containers are not mentioned, which can provide attack surface area for Spectre exploits [34].

4.3. FaaS – Function as a Service

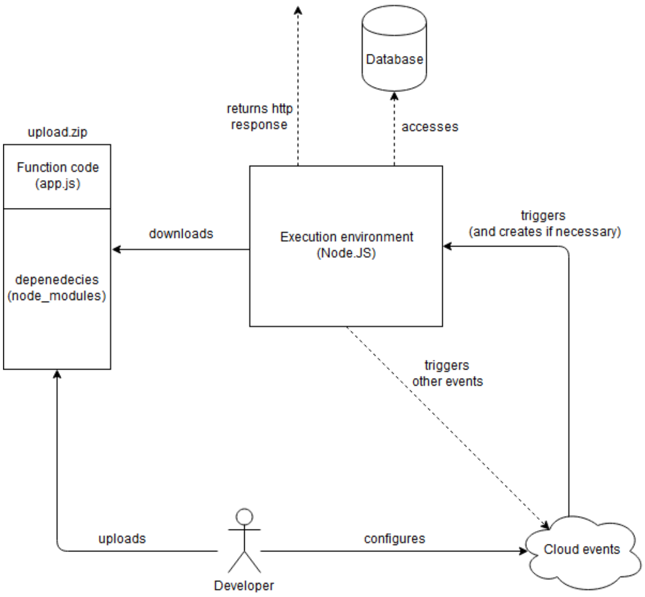

Figure 4. FaaS example for a Node.JS application.

FaaS further abstracts service infrastructure. Conceptually a developer writes a function, uploads it to the FaaS service and connects events like HTTP requests or database events as an input source.

When an event occurs the function is called with the event's parameters and, in the case of an HTTP request, builds and returns a API response. The benefits are automatic scaling and a payment model where the customer only pays for the function's execution time [7].

In reality a "function" is a package of code, configuration and module/library files. Therefore, even if the service for example only allows Node.JS execution this cannot be considered a security feature as a node module can contain native, compiled binary files [6].

Due to the high abstraction level, internal implementation details are sparsely documented. AWS Lambda notes that Lambda functions are isolated using EC2 techniques, i.e. hypervisor isolation [6]. Google Cloud Functions (GCF) in contrast only states that functions are run in "its own isolated secure execution context" [14]. Wang et al. [41] further investigates FaaS implementations.

If a FaaS would only use docker to isolate functions of different customers, the risk of a speculative attack would be higher compared to a Docker PaaS provider. Since function are based on small number of available runtime environments (e.g. GCF: Go, Python and Node.js 6,8,10) [14] an attacker has implicit knowledge about the victims shared library and executable files.

Therefore, an attacker could not only attempt co-residency to gain access to a specific target's secrets, but instead mount a broader attack acquiring data from random other tenants. Similar to spam phishing emails, the goal would not be one large "heist" but instead gaining value from many smaller thefts.

5. Prevention and detection

5.1. Hardware, Software and Compiler

While meltdown could be mitigate by operating system patches [2], the Spectre vulnerabilities have required cooperative changes to hardware, compilers and vulnerable software itself.

On the hardware side, some vulnerabilities have been removed in newer processors, others can be circumvented through new instructions that allow explicit prevention of speculative execution [36].

Compilers have been adapted to reduce the generation of code blocks that are vulnerable to some forms of Spectre attack [1] and insert code that traps speculative execution in an endless loop [34].

All of these changes require recompilation of programs, libraries, operating systems and drivers. Additionally all of these mitigation have some form of performance drawback and are therefore not enabled by default.

5.2. In-memory encryption

To prevent a Spectre attack from accessing secret data, the data could be encrypted in-memory. This technique however, has two weaknesses. The first issue arises from the fact that to make use of the encrypted data the process has to, albeit briefly, decrypt the data. This results in the data still being vulnerable, but reduces an attacker's window of opportunity to access the secret.

The second issue stems from encryption key storage. Since the key itself has to be stored in-memory for encryption/decryption it could be stolen together with the encrypted data.

In a shared kernel environment, e.g. Docker, the key can be hidden completely from the attacker by storing it outside of the victims address space. This is due to the limitation of Spectre, that an attack can only access memory the victim has permission to [28].

Therefore, when a program uses functionality like Windows' `CryptProtectMemory` [16] the key is stored in a different processes memory or even kernel space memory.

In the case of a cross VM attack in a hypervisor environment, the attacker gains access to all VM memory. This only raises the difficulty of the attack since the attacker has to know the location of both the encrypted secret and the decryption key.

5.3. Heuristic detection

By using CPU caches as a side-channel a Spectre attack can create a distinctive cache usage pattern which could be used to detected ongoing attacks [18]. This is due to Spectre deliberately causing cache misses to flush the cache or cause speculative execution by delaying memory access. Detection in this way has the benefit of being easy to deploy to existing system since no changes to vulnerable software is required. [18, 13]

However since Spectre does not rely on a specific side-channel for information access [28] detection could be circumvented. Gruss et al. [22] even demonstrates a different cache side-channel that would not be detected by monitoring process cache-misses.

Additionally by simply slowing down a Spectre attack it could also circumvent detection [13].

6. Conclusion

Without any definite resolution of the processor inherent flaws in the near future, Spectre exploits can paint a bleak picture for the future of secure computer systems. However, just like its folklore counterpart, Spectre seems to have spread terror not entirely proportional to its real threat (a CVSS score of 5.6 out of 10) [31].

Today most data breaches are still possible through simple software flaws like SQL injections [29], passwords stored in plain text [20] and even internet facing databases without any access control [32].

But for customers for whom data security is top priority, all possible approaches have to be considered. However, for such applications cloud providers already offer solutions where servers are not shared with other customers [17].

On the other hand many important discussions have begun because of the broad interest in Spectre. Processor manufactures, especially Intel though arguably disproportionately, lost costumer's trust in the security of their products, potentially opening a new market segment for slower but more secure processors to competitors. It also raised awareness of side-channels, changing the perspective of secure systems.

Lastly, during the making of this work it became evident that cloud platform should be more transparent about their systems. Documentation should be explicit about how tenants are isolated, i.e. through containerization with docker or with a hypervisor, to allow customers to assert the risk of hardware side-channel-attacks for their systems.

Abbreviations

ALU	arithmetic logic unit
AWS	Amazon Web Services
CVSS	Common Vulnerability Scoring System
FaaS	Function as a Service
GCF	Google Cloud Functions
IaaS	Infrastructure as a Service
IPO	input-process-output
MMU	memory management unit
NIST	National Institute of Standards and Technology
PaaS	Platform as a Service
SaaS	Software as a Service

References

[1] */Qspectre — Microsoft Docs*. 2018. URL: https://docs.microsoft.com/en-us/cpp/build/reference/qspectre?view=vs-2019.

[2] *15. Page Table Isolation (PTI) — The Linux Kernel documentation*. 2019. URL: https://www.kernel.org/doc/html/latest/x86/pti.html.

[3] *7-Zip LZMA Benchmark — Intel Skylake*. 2019. URL: https://www.7-cpu.com/cpu/Skylake.html.

[4] *AWS Fargate on Amazon ECS - Amazon Elastic Container Service*. 2019. URL: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html.

[5] *AWS Fargate Pricing - Run containers without having to manage servers or clusters*. 2019. URL: https://aws.amazon.com/fargate/pricing/.

[6] *AWS Lambda – FAQs*. 2019. URL: https://aws.amazon.com/lambda/faqs/.

[7] *AWS Lambda – Product Features*. 2019. URL: https://aws.amazon.com/lambda/features/.

[8] *Azure Container Instances — Microsoft Azure*. 2019. URL: https://azure.microsoft.com/en-us/services/container-instances/.

[9] Daniel J Bernstein. “Cache-timing attacks on AES”. In: (2005).

[10] David Bernstein. “Containers and cloud: From lxc to docker to kubernetes”. In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84.

[11] *Best practices for writing Dockerfiles — Docker Documentation*. 2019. URL: https://docs.docker.com/develop/develop-images/dockerfile_best-practices.

[12] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management.* ” O’Reilly Media, Inc.”, 2005.

[13] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. “Real time detection of cache-based side-channel attacks using hardware performance counters”. In: *Applied Soft Computing* 49 (2016), pp. 1162–1174.

[14] *Cloud Functions Execution Environment*. 2019. URL: https://cloud.google.com/functions/docs/concepts/exec.

[15] *Container Hosts and Multi-tenancy — Container Security Guide — OpenShift Container Platform 3.5*. 2019. URL: https://docs.openshift.com/container-platform/3.5/security/hosts_multitenancy.html.

[16] *CryptProtectMemory function (dpapi.h) — Microsoft Docs*. 2019. URL: https://docs.microsoft.com/en-us/windows/desktop/api/dpapi/nf-dpapi-cryptprotectmemory.

[17] *Dedicated Instances - Amazon Elastic Compute Cloud*. 2019. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/dedicated-instance.html.

[18] Jonas Depoix and Philipp Altmeyer. *Detecting Spectre Attacks by identifying Cache Side-ChannelAttacks using Machine Learning*. 2018. URL: https://www.betriebssysteme.org/wp-content/uploads/2018/10/WAMOS_2018_paper_12.pdf.

[19] *EC2 Instance Pricing – Amazon Web Services (AWS)*. 2019. URL: https://aws.amazon.com/ec2/pricing/on-demand/.

[20] *Facebook Stored Millions of Passwords in Plaintext—Change Yours Now*. 2019. URL: https://www.wired.com/story/facebook-passwords-plaintext-change-yours/.

[21] Agner Fog. “The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers”. In: *Copenhagen University College of Engineering* (2012). URL: https://www.agner.org/optimize/microarchitecture.pdf.

[22] Daniel Gruss et al. “Flush+ Flush: a fast and stealthy cache attack”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2016, pp. 279–299.

[23] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[24] *Heroku FAQ: Isolation and security*. 2019. URL: https://devcenter.heroku.com/articles/dynos#isolation-and-security.

[25] Mehmet Sinan Inci et al. “Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud.” In: *IACR Cryptology ePrint Archive* 2015.1-15 (2015).

[26] Paul C Kocher. “Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems”. In: *Annual International Cryptology Conference*. Springer. 1996, pp. 104–113.

[27] Paul Kocher, Joshua Jaffe, and Benjamin Jun. “Differential power analysis”. In: *Annual International Cryptology Conference*. Springer. 1999, pp. 388–397.

[28] Paul Kocher et al. “Spectre attacks: Exploiting speculative execution”. In: *arXiv preprint arXiv:1801.01203* (2018).

[29] *Magento 2.3.1, 2.2.8 and 2.1.17 Security Update — Magento*. 2019. URL: https://magento.com/security/patches/magento-2.3.1-2.2.8-and-2.1.17-security-update.

[30] Peter Mell, Tim Grance, et al. *The NIST definition of cloud computing*. 2011.

[31] *NVD - CVE-2017-5753 (Spectre)*. 2019. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-5753.

[32] *Over 12,000 MongoDB Databases Deleted by Unistellar Attackers*. 2019. URL: https://www.bleepingcomputer.com/news/security/over-12-

000-mongodb-databases-deleted-by-unistellar-attackers/.

[33] *Overlay Filesystem*. 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/overlayfs.txt.

[34] *Retpoline: a software construct for preventing branch-target-injection*. 2018. URL: https://support.google.com/faqs/answer/7625886.

[35] Thomas Ristenpart et al. “Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds”. In: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM. 2009, pp. 199–212.

[36] *Speculative Execution Side Channel Mitigations*. 2018. URL: https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf.

[37] Andrew S Tanenbaum and Albert S Woodhull. *Operating systems: design and implementation*. Vol. 68. Prentice Hall Englewood Cliffs, 1997.

[38] *Vmware ≫ Esxi : Security Vulnerabilities*. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-252/product_id-22134/Vmware-Esxi.html.

[39] *VMware response to ‘L1 Terminal Fault - VMM’ (L1TF - VMM) Speculative-Execution vulnerability*. 2019. URL: https://kb.vmware.com/s/article/55806.

[40] *VMware Response to Speculative Execution security issues (Spectre and Meltdown)*. 2018. URL: https://kb.vmware.com/s/article/52245.

[41] Liang Wang et al. “Peeking behind the curtains of serverless platforms”. In: *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 2018, pp. 133–146.

[42] Yinqian Zhang et al. “Cross-tenant side-channel attacks in PaaS clouds”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 990–1003.