Advanced Topics in Network and System Security: ZombieLoad

Tomi Jerenko
jerenkotomi@gmail.com
Brandenburg University of Technology Cottbus-Senftenberg
Cottbus, Germany

ABSTRACT

Meltdown and Spectre have put the security aspect of modern CPUs under a spotlight by showing that due to performance optimizations vulnerabilities may exist in the hardware. While Spectre exploits branch prediction and speculative execution, Meltdown focuses on the transient execution of instructions and fault handling. Following the principles of the two, new fault-driven transient-execution and speculative execution attacks are emerging.

In this paper, the Zombieload attack, based on the Zombieload paper[16], is presented. By exploiting Intel CPUs' behavior of speculatively matching instruction pointers in the Line-Fill Buffer (LFB) to transient instructions, the attack is able to leak data across privilege boundaries and across threads on the same physical core. Different possible leakage flows and multiple attack variants to leak data are presented.

This paper's contributions are richer summaries of background topics necessary to understand the Zombieload, and an overview of Zombieload related works and their comparison to Zombieload. Additionally, the Zombieload attack is presented in a simplified version of the original paper. Summaries of its attack variants, application possibilities, evaluation and countermeasures are presented.

KEYWORDS

Transient execution, microarchitectural data sampling, meltdown, line-fill buffer, side-channel attack

1 INTRODUCTION

The CPU is one of the main components of computers, through which all data is processed and controlled. For that reason, it must be secure and reliable. For many years, CPU manufacturers have focused primarily on performance aspect of hardware development, causing security aspect to be neglected. Meltdown [14] and Spectre [13] showed, that performance optimizations might have security implications, i.e. leaving the hardware vulnerable. An examples of these performance optimizations are speculative execution and out-of-order execution.

The speculative execution principle is used by the CPU to try to use execution units to keep them busy from idling. Instead of stalling them by waiting for values to arrive from the slower memory, the CPU saves a checkpoint and starts executing other instructions by making an educated guess of a program flow. In that case, when the initial value arrives from the memory and the already executed instructions are found to be incorrect, the CPU simply reverts its state to the saved checkpoint. This causes an additional overhead, however, such optimization increases the system's overall performance significantly.

Out-of-order execution of instructions is another performance optimization which lets the CPU to execute data independent instructions out of order. Firstly, the CPU breaks down the program's instruction sequence into micro operations (μOPs) which can be executed independently. Their independence makes their execution order irrelevant, as it does not have any effect on the final result. When all of them are executed, the CPU reconstructs them into their original order and applies them to the architectural state accordingly.

Spectre and Meltdown exploit the speculative and out-of-order execution of instructions respectively. While spectre relies on tricking the branch prediction unit to speculatively execute instructions, the basis for Meltdown is transient execution of instructions. Both compete in a race against the CPU in which they want the CPU to execute some instructions before it discovers they do not belong into the correct program flow, thus discarding them. These instructions do not cause any traces to the architectural state, however, they leave traces in the microarchitectural elements of the CPU, e.g. cache. These changes can be observed by a cache side-channel attacks such as FLUSH+RELOAD [18].

The Zombieload [16] goes even further and extends the transient execution, demonstrated by Meltdown, to target the microarchitectural elements themselves. As these elements do not distinguish between privileges, the Zombieload can leak any data currently residing in them - specifically, it targets data in the line-fill buffer (LFB). The in-flight data in the LFB is arbitrary, i.e., it can be any data that the CPU is currently processing. Therefore, an attacker cannot control which data gets loaded into the LFB. It was shown by the Zombieload, that when an page fault occurs due to the memory read, the CPU speculatively matches a value from the LFB. It executes instructions with wrong values, hoping that the matched value was the correct one. When it finds out a mistake was made, their results are discarded. This leads to a observable cache state changes which FLUSH+RELOAD[18] is able to leak. The leaked data has to be reconstructed and interpreted by the attacker, thus the attack is also classified as a microarchitectural data sampling attack (MDS).

Outline: Section 2 gives a background information of elements used later on. In Section 3 overview of attacks related to Zombieload is given. Section 4 describes the attack and its pillars. Section 5 gives overall results of the attack and its specific application results. Section 6 discusses possible countermeasures and their effects. Section 7 concludes the paper.

2 BACKGROUND

This section gives provides the background information necessary to understand the Zombieload [16].

2.1 Memory Management

The process of memory management deals with the organization, allocation and control of memory resources used by computer ran programs. Physical addresses, used to address memory cells in chips (e.g. RAM), are mapped/translated to virtual addresses by the memory management unit (MMU). The main benefits of this abstraction are memory isolation of processes and a feeling of contiguous access of infinite RAM addresses.

For efficiency purposes, virtual addresses and physical addresses are grouped into fixed-length blocks called pages and page frames respectively. Pages represent data, whereas page frames represent a physical area to store that data. Kernel handled data structures, called page tables, contain page to page frame mappings and are held in the main memory [8]. Page tables can be swapped from the memory to a disk to free up the memory, and swapped from the disk to the memory to be reused.

Intel processors' MMU handles 4 KB large pages. As shown in Figure 1, it divides virtual address' bits into multiple fields to be used for virtual to physical translation, called multi-level page translation. Virtual address' 10 most significant bits reference page table address within the directory, 10 middle bits reference the page address in the memory and 12 least significant bits represent the data's relative position within the page. A special register in CPU holds a pointer to a directory. On context switch, the register's pointer value is changed [8].

Page table entries also include fields (bits) such as accessed, dirty and privilege. Accessed bit of an entry is set whenever the MMU addresses its page frame, and the dirty bit is set every time a write operation to the page frame is pending. Both are set by the hardware but neither is reset by it, instead the operating system must do it. Privilege bit holds privilege information — whether the address is considered to be trusted by kernel or not — and is checked by the MMU. In addition, the MMU communicates with cache called translation lookaside buffer (TLB) which is used to temporarily store virtual to physical address maps. Accessing the TLB is faster compared to the main memory, thus serves the purpose of speeding up the translation of frequently used addresses [8].

To isolate memory between processes, the Linux operating system distinguishes between two memory regions: user space and kernel space. Every program has an access to its user space memory and to some kernel functions through system calls. Whenever the CPU is running in privileged mode, the kernel space can be accessed. Kernel's role is to switch the CPU's mode between privileged and non privileged, and to restrict user processes to use memory outside of the user space by setting the privilege bit of an address — bit 0 for kernel space address and bit 1 for user space address. In contrast, kernel applications run in kernel space and are able to access the whole physical memory and hardware — they have absolute control over our machine. If a kernel space memory is accessed while the CPU is not in kernel mode, a page fault occurs [14].

Sometimes pages are shared among processes for inter-process communication or to reduce memory footprint of identical contents. For the latter purpose, two types exist: content-aware and content-based page sharing. In the first, the identical pages are identified by their contents' disk locations; used when using shared libraries or executing the same text segment in different processes. In the case of second type, the system is actively scanning active memory for identical contents and merging them. To protect the shared memory pages from malicious modifications, a system uses copy-on-write technique. The system lets the processes read the pages freely; only in the case of writing, the pages are firstly copied and mapped into the process' own address space, and then written to [18].

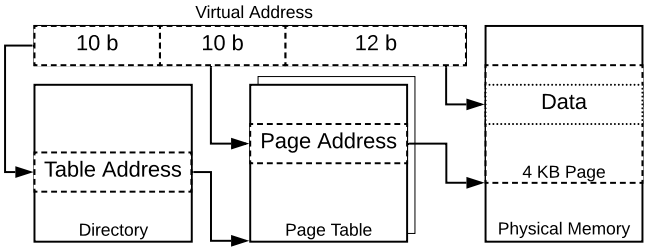

Figure 1: Virtual to Physical Address Translation [8].

2.2 Caches and Cache Architecture

When talking about caches it is useful to know why they are required. There exists an inverse correlation between read/write speed to memory and effort to persist the data. To cover both aspects, modern computers implement a system/hierarchy of cooperating memory units to serve the data fast and persist it for as long as possible.

On the lowest level are storage units such as solid state drives (SSD). They are considered to be extremely slow compared to how fast the CPU can process data, therefore useless to fetch instructions from directly (however they are very good at persisting large amount of data for long periods of time at minimal energy consumption). For this purpose the next level of memory called main memory is used.

The data served from this level is served considerably faster however, compared to the speed of CPU, still slow. The data persists for as long as the power to the unit is supplied - after that it starts discharging. Main memory is made out of dynamic random allocated memory (DRAM) cells; a single DRAM cell is made out of one transistor and one capacitor. When interacting with the cell the transistor lifts the gate and capacitor needs to charge or discharge to be read, or written to respectively. Charge/discharge of capacitor is not instant, introducing delay. Another factors to delay data retrieval is the size of main memory and its physical location — it is relatively large and further away from CPU. Compared to the next level, static random allocated memory (SRAM), DRAM is cheap to make and consumes less power [3].

Next level are CPU caches. They are made out of SRAM cells and are much smaller than main memory. They are positioned close to CPU. Generally, a single SRAM cell is made out of six transistors. They require constant power supply and consume more power than DRAM cells, and are more costly to produce. Reads and writes to them are instant since charging and discharge is not required [3]. The main purpose of these caches is to act as a temporary storage for recently accessed values from the main memory. If the CPU needs to subsequently retrieve these values again, it retrieves them from the cache, thus saving time and reducing the delay caused by the main memory accesses [18].

To increase the speed of data retrieval even more, a multi-level cache hierarchy is used in modern processors. Normally three levels are used: L1, L2 and L3. Each CPUs physical core uses its own L1 and L2 cache, whereas L3 cache is shared uniformly between all cores. Cache L1 is the smallest but closest to the CPU core, and cache L3 is the largest but furthest. Cache's unit of memory is called cache line and is typically 64 bytes large. Cache L1 is additionally split into instruction cache and data cache, also called L1-I and L1-D respectively. L3 or Last-Level Cache (LLC) is inclusive, meaning it includes copies of all data stored in upper cache levels. When a cache line is flushed from the LLC it is also flushed from other cache levels to retain data consistency [18]. A single cache has multiple cache lines, depending on the design of CPU. A single trip to the main memory retrieves, due to performance reasons, one cache line size of data at once and replicates it throughout cache hierarchy. The frequently used data is cached on levels closer to the CPU and is pushed down the hierarchy when it's not used so frequently.

Cache hits and misses: when the CPU requests to read from or write data to the main memory it checks the whole cache hierarchy from L1 to L3 if they contain the data it needs. If the CPU finds the data inside of the cache it is considered a cache hit. This means that it can use the data much faster than waiting to get it from the main memory. On the other side, if the CPU looks the data up in the cache and doesn't find it, it is considered a cache miss. In the latter case the delay is even longer to retrieve data from the main memory. However by using caching system we observe 91.5% performance gain overall [7].

2.3 Out-of-order Execution

Some CPUs use out-of-order execution concept to optimally execute instructions to prevent their execution units from idling. For CPU's components to be used effectively in a timely manner, instructions are executed immediately after their resources are available as long as it doesn't impair the final output of an instruction sequence, meaning they have to be independent from one another. Additionally, speculative execution of instructions extends this concept. In the case of speculative execution, the CPU saves its state and predictively executes instructions which might be needed later. It commits their effects the moment it is sure they are needed, the state is reverted to the saved one and results are disposed of. A typical example is branch execution, where branch code is executed and it's conditions are checked later on. The temporal state, in which the instructions are executed yet not committed, is called micro-architectural state. The moment the instructions are committed, the micro-architectural state becomes architectural, i.e., the consequences of operations are visible to the outside world [14].

Branch prediction units are CPU's components used to guess which branch should be speculatively executed. There exist multiple different approaches e.g. static, dynamic and n-level branch prediction. First one is to make a decision based only on observation of a single instruction, second one uses statistics at run-time and third one is based on the previously executed branch count [14].

2.4 Line-Fill, Load, and Store Buffers

In addition to whole memory hierarchy and CPU cache hierarchy, Intel's CPUs go even further and implement yet another performance optimizations called line-fill buffer, load buffer and store buffer. As shown in Figure 2, they are positioned between CPUs L1 cache and execution units.

Load and store buffers serve as a queue for loads and writes respectively. When the CPU loads or writes data to an address it enqueues it to these buffers; for stores the corresponding written data is also enqueued. Loads and stores from these buffers happen asynchronously while the CPU is processing other instructions. If a load happens immediately after write to the same address, i.e. before the data from store buffer was written to main memory, the unsaved data from this dirty address is directly forwarded from store buffer to load buffer (this is called store-to-load forwarding)[9].

Intel's processors use so called Line Fill Buffers (LFBs) in collaboration with load and store buffers, L1d cache, L2 cache and execution units. They keep track of unresolved memory requests, therefore allowing the CPUs to aggressively speculate execution of micro operations (μOPs) without any awareness of the virtual or physical addresses involved. They provide the following functionalities [17]:

- ***Non-blocking caches:*** In the case of cache misses, the CPU spends more time to find virtual to physical map compared to cache hits, as it must look into the upper level caches or even into the main memory. This causes a cache to be blocked until the translation is complete. The LFB temporarily stores unresolved load requests to track their physical addresses until the data is available.
- ***Load squashing:*** Multiple load misses with the same physical address are merged.
- ***Write combining:*** Multiple store operations to the same cache line are merged into a single LFB entry before applying the final result to the memory hierarchy.
- ***Non-temporal requests:*** Uncacheable loads and stores are done exclusively through the LFB.

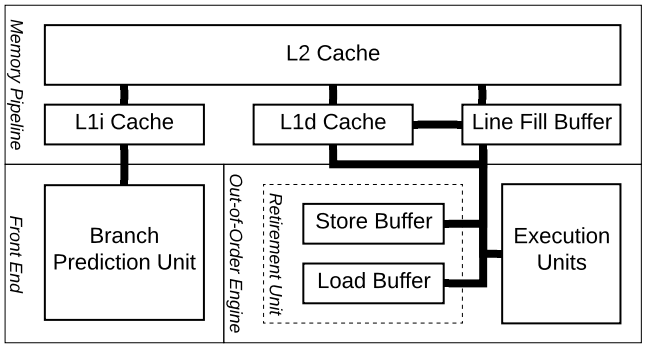

Figure 2: Simplified overview of the Intel Skylake architecture based on the RIDL paper [17].

2.5 Processor Extensions

Microcodes are higher level instructions consisting of multiple lower level (hardware) instructions. This abstraction makes updating, upgrading and adding new functionalities easier. Microcode is also used for performing complex operations such as fault handling or page table modifications [16].

Intel TSX is an instruction set extension for transactional code execution. If the code is executed successfully, other logical processors see it as an atomical commit. When an error occurs during the transaction, the CPU's state is returned to the one before the transaction started [16]. When a read or write instruction is issued inside of a transaction, it is considered to be a part of the transaction's write-set. Intel defines read and write sets as sets of instructions performed within the TSX transactions. These sets are also kept in each L1 cache line. If another logical CPU tries to read or write to the read or write set, a data conflict occurs and the transaction is aborted [2].

Intel SGX is an instruction set extension for isolated execution of trusted code. Data and code operating on the data are stored in hardware protected memory area called enclave. The hardware ensures that all other sources except the authenticated application using the SGX can not access it. The enclave contents are encrypted and are only decrypted in CPU. Whenever the CPU is not in enclave mode and tries to access the Intel SGX reserved memory, a microcode assist is triggered. In the case of load instruction the value '0xFF' is returned, whereas write instructions are silently ignored [16].

Figure 3 represents a model for secure remote computation which Intel SGX is based on. In fact, the SGX relies on software attestation. At the first step towards secure remote computation, a setup phase ensures that the data and its code is loaded into the enclave. After a successful setup, enclaves are marked as initialized and hash value is calculated over their contents. Furthermore, a digital signature is generated over enclave's contents which are signed by Intel's private key stored in the hardware [16]. A remote party requests so called software attestation procedure to be sure it is going to be sending its code and data to the correct enclave. Once the setup procedure is finished and remote party is sure of its enclave, exchange of further code with data can take place. The third party enters enclaves with the *EENTER* instruction and the CPU goes into enclave mode. After the execution of tasks is finished, the enclave exits with the *EEXIT* instruction. In case of an hardware exception occurring during enclave instructions are executing, the enclave state is saved within the enclave, and the CPU goes out of the enclave mode. The execution of instructions can be resumed at a later point in time using the *ERESUME* instruction [10].

To generate keys for the purposes of SGX software attestation, anonymity and membership revocation Enhanced Privacy Identification (EPID) implementation of the ISO 20008 standard is used [1].

2.6 Attacks

In this section the underlying attacks used in Zombieload are explained.

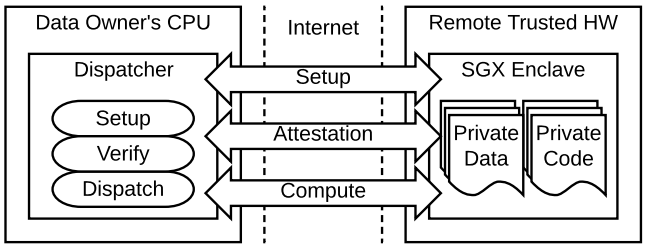

Figure 3: Trusted Computing Overview [10]

Cache attacks are types of attacks that exploit timing differences of cache accesses vs. memory accesses. Because of the page sharing mechanism, it is possible to determine the addresses of the pages being shared. When a process accesses a memory location, virtual to physical address is translated and saved in the cache. Consequently an attacker can target these addresses to flush them out of the cache and later observe whether some process accessed any of them. Because accessing the cached data is much faster than accessing the data in the main memory, the attacker can measure if the address has been accessed by another source between he flushed the address and read from it. One of such attacks is called FLUSH+RELOAD, which works on a single cache line granularity and targets the LLC. Because the LLC is inclusive, removing one of its cache lines means removing all replicated cache lines in other cache levels as well (caches L1 and L2). The combination of instructions *clflush* and *reload* is used to flush cache line and reload the address respectfully. Time is measured by reading the CPU's *Time Stamp Counter (TSC)* register using *RDTSC* instruction [18].

Transient execution attacks are types of attacks that exploit the out-of-order execution and speculative execution performance optimizations. μOPs which are executed speculatively and in an out-of-order fashion, but their results are never visible architecturally, are called transient instructions. Despite the fact that these kinds of instructions have no effect on a CPU's architectural state, they do leave observable microarchitectural state changes (e.g. cache state changes) observable from the outside world. Attacks that exploit this performance optimization behavior are called transient execution attacks [16].

3 RELATED WORKS

This section provides an overview of Zombieload related attacks and their classification.

3.1 Meltdown

Overview: Meltdown is a side-channel transient instruction execution attack which is able to leak kernel memory from user space. General attacker model is a setup, where an attacker can run any unprivileged code with the privileges of a normal user on the attacked system. The code tries to access user inaccessible addresses to get the secret data. Due to the out-of-order execution of operations, the code targeting inaccessible addresses still gets executed in a transient way. Furthermore, transient instructions leave observable changes in caches, which are later used as a covert channel for the

attacker to leak hidden data. Access to cache lines is timed by the attacker to determine whether a value has been cached or not. If the value was cached, its index directly corresponds to the secret value stored in the attacker specified virtual address [14].

Attack: First step for the attack is to get arbitrary data, otherwise inaccessible by the attacker, loaded into a register referenced by a virtual address. Because hardware implemented permission checking by the CPU is considered safe, operating systems usually map the entire kernel into the virtual address space of every user process. If a process tries to access its kernel memory from the user space, the CPU blocks this access and raises an exception which results in a microcode assist. Because of the out-of-order execution, some illegal memory accesses can still happen before the exception is finally handled. Their results are later discarded architecturally, yet leaving visible changes microarchitecturally (in cache) [14].

In the second step, the goal is to execute transient instruction which accesses illegal memory location (secret) and leaves observable microarchitectural change after the result of the instruction has been discarded. Furthermore, an empty uncached array of length 256×4096 bytes is allocated into the memory (it is assumed that the system uses 4 KB pages). An index of the array is then calculated by multiplying the value of the secret data, residing at the illegal address, with 4096 and tried to be accessed. If this is done in a transient way, the array is cached at the specified index corresponding to the secret. The change remains in the cache even after the exception is handled and operations discarded [14].

The third step is to utilize the covert channel and retrieve the secret to the architectural state. For this purpose FLUSH+RELOAD is used, to retrieve the secret from the cache. If the step 2 was successful, the CPU cached one line of the probe array. Furthermore, 256 lines of the array are iterated through and their access times measured. If much shorter access time to the array entry is detected, its index directly corresponds to the leaked secret value [14].

Mitigation: Meltdown is mitigated by KAISER [11], a kernel modification, which prevents mapping kernel to user space. Nonetheless, there still are some privileged memory locations being mapped to the user space. These locations do not contain any secret data, but they might contain certain pointers. Therefore, KAISER effectively mitigates Meltdown, but not fully. The simplest solution would be to disable out-of-order execution, but this would reduce the performance of modern CPUs greatly. A proposed solution is to implement hard-split of the memory by adding an additional hard-split bit in the control register. The bit would tell the memory fetch that the kernel is residing in the upper part of the address space and user space in the lower part of the address space [14].

3.2 Spectre

Overview: Spectre is a side-channel speculative execution attack which exploits the speculative out-of-order execution of instructions by branch misprediction. It is able to exploit process isolation boundaries (i.e. leak secrets from process to process) and sandboxing (i.e. leak secrets within a process). In general, it aims to trick the CPU to speculatively execute instruction sequences, which temporarily violate program's semantics. This is done by mistraining the branch predictors to mispredict the program flow for conditional branches or indirect branches. The mispredicted program flow leaves observable changes in caches, which are later observed by cache side-channel attacks (e.g. FLUSH+RELOAD [18]), resulting in secret information leakage [13].

Attack: There are two variants to the attack. Variant one, called Exploiting Conditional Branch Misprediction, can be used to exploit sandboxing (e.g. when a website executes JavaScript code, browser secrets, such as passwords, can be leaked). Variant two, called Poisoning Indirect Branches, is used to exploit process isolation boundaries (e.g. attacker tricks the victim to execute some vulnerable code called gadget, which leaks victims secrets to cache). The main starting point for both variants is to mistrain the branch prediction unit, which causes the CPU to wrongly assume which instructions to execute out-of-order. In the case of the variant one, this is done for conditional branches within a single process to access out of bounds values of an array. In the case of variant two, one process causes a bias for prediction of indirect branch flow to an incorrect address, which causes the second process to execute the code residing in that address [13].

To exploit conditional branches (variant 1), the attacker's program repetitively executes valid conditional branch flow with attacker controlled logical branch conditions. The values for conditions should be accessed from the main memory and not cache. The CPU remembers the constant flow within the branch and predicts that the subsequent code will execute in the same way. This allows the CPU to execute instructions in advance before the conditional statement is resolved. Attacker then suddenly tries to access an out of bounds value of an array in the usual branch flow (the out of bounds value should be cached). The CPU speculatively executes the instructions that perform some operations with the secret value, only to later find out that it should not have done that. The executed instructions are then discarded, and not allowed to be applied to the architectural state. In any case, as these instructions were once already executed, they have left the cache state to be modified [13].

To exploit the indirect branches (variant 2), the attacker must trick the victim to execute a gadget (a code fragment acting as a covert channel) in victim's address space speculatively. Firstly, the attacker must locate a gadget within victim's executable memory location (e.g. shared libraries). Secondly, after finding the suitable gadget's address, the attacker massively executes indirect branch flows to that address. The pointers are being remembered by the Branch Target Buffer (BTB), causing it to be flooded by them. When the victim executes an indirect branch in its code, the program flow is fed a pointer from the poisoned BTB, causing the CPU to speculatively, on victims behalf, execute instructions residing in that address. This flow is later found out to be incorrect and therefore reverted, yet leaving cache state changes [13].

The final phase of the Spectre attack is to extract the secrets from cache. For this purpose, FLUSH+RELOAD [18] cache attack can be used.

Mitigation: as the speculative execution of instructions is crucial for Spectre attacks, disabling it would fully stop this kind of attacks. Practically, this is not entirely possible as this would cause the CPU to perform considerably slower, therefore the authors

propose that a setting which lets a software to disable speculative execution should be introduced. Additionally, to prevent exploiting sandboxes splitting the running code in separate processes is proposed (refers to variant 1). In the case of indirect branch poisoning, the Indirect Branch Restricted Speculation (IBRS), Single Thread Indirect Branch Prediction (STIBP) and Indirect Branch Predictor Barrier (IBP) are used to prevent indirect branches in privileged code to be affected by less privileged code, branch prediction sharing between the software hyper threads on the same core, and software before set barrier to affect branch prediction of a software after the barrier respectively [13].

3.3 RIDL

Overview: RIDL [17] is a Microarchitectural Data Sampling (MDS) attack focusing on leaking data from Line Fill Buffer. Since the CPU performs load and store operations through the LFB, the attacker is able to leak any arbitrary data that is passing through the LFB during the attack execution time frame (i.e. it is possible to leak data from any source). Directly selecting the data that passes through the Line Fill Buffer is not possible, therefore the attacker can only observe it. Because RIDL relies on the fact that the targeted secret data is in-flight during the attack (i.e. the secret data must be present in processor when the attack is being executed), other means should be used to trick the adversary to get the data in-flight. Furthermore, when the data is leaked to the cache, attack FLUSH+RELOAD is used to extract the secret values speculatively passed to the cache. Lastly, because the leaked data is arbitrary, means to effectively eliminate the noise, by eliminating irrelevant data and reconstructing the relevant data, must be used (e.g. synchronizing with the victim) [17].

Attack: RIDL uses a very straightforward straight-line code without any branch predictions, invalid accesses or error suppression. Firstly, an empty buffer array is prepared to serve as a covert channel. Secondly, the code accesses some uncached valid memory address, thus the CPU must fetch its value from the main memory. Because the read is slow, the CPU already loads speculated in-flight value from the LFB, hoping it is the correct one. When the correct value from the original instruction gets returned, the CPU detects a mistake and reverts any kind of modifications done by the instructions. In the time frame between receiving a speculated value from the LFB to mistake detection, instructions accessing an empty buffer array have to be executed. These instructions use the in-flight value to access an index in the empty buffer array. As a result, the array line with the index (the speculated value) is cached. Because the CPU reissues the instruction due to the mistake, it repeats the memory access twice. Therefore, two cached values are expected (the actual value and the secret value). Buffer array indexes are then accessed to see which ones are accessed faster. The fast accessed indexes correspond to either the secret or some memory value [17].

When the data is being leaked it is required to be synchronized with the victim and to determine how the leaked bytes belong together sequentially. The most straightforward approach is called hard synchronization. The goal is to repetitively leak the same bytes and to compare them to the already leaked ones. For this purpose, the attack called mask-sub-rotate is used to allow the attacker to filter out noise and to leak new secret bytes. The attack is based on the fact that the parts of the secret bytes are already known. In the process, a combination of masking, subtraction and rotation of the known bytes with freshly leaked bytes is used. This allows an attacker to determine whether the freshly leaked bytes are the ones he is looking for or not [17].

Additionally, to get the secrets in-flight, i.e. into the LFB, different tricks are used. One such example is in the case of cross-VM attacks where the attacker keeps trying to authenticate through a SSH from one VM to the victim VM. This causes the privileged process to open the /etc/shadow file containing secrets and, consequently, loading it into the TLB. Another example is to leak kernel pointers and other kernel data by executing a system call and mounting the attack right after [17].

Mitigation: as the RIDL relies on speculative and out-of-order execution of instructions, disabling these primitives prevents the attack completely. Because this is a hardware issue and therefore cannot be applied in practice quickly, software mitigations are deployed. It is proposed simultaneous multi-threading to be disabled. Additionally, Intel points out that the RIDL can be mitigated by ensuring only trusted code is running in the sibling thread. This seems to be a very complex and hard task because it requires scheduler modifications and synchronization at system call entry points. In any case, it is still insufficient to protect SGX enclaves, sandboxes or leaking within single thread. For this purpose, Intel published update for microcode that allows LFBs, load ports and store buffers to be flushed. This is done with *verw* instruction, which has to be used in combination with speculation barriers such as *lfence* [17].

3.4 Fallout

Overview: Fallout is a MDS attack focusing on leaking data from store buffer. In contrast to RIDL, it specializes in leaking pending writes to main memory, i.e, the writes must be in-flight and leaked addresses can not be directly controlled. After executing write to a memory location, CPU enqueues the write data and its address to the store buffer, which in turn holds the value until it is finally stored to the main memory. Because the push to main memory happens asynchronously, consequent loads firstly check the store buffer for pending writes. If it contains the data, it is fetched from there. An attacker exploits this behaviour during transient execution of instructions. This algorithm is called write transient forwarding (WTF) and is patented by Intel. In a case the address translation of some μOP fails, the WTF algorithm will forward pending write data from the store buffer to load instructions if some lower bits of their addresses match. Fallout includes additional techniques that allow an attacker to check which virtual addresses are mapped to physical addresses — as a consequence this is also able to break ASLR. It builds on the logic of Spectre [13] attack and expands it to the point where it is able to leak writes performed by kernel [9].

Attack: The attack comes in four parts, each adding new functionality the previous one: write transient forwarding, data bounce, fetch+bounce, speculative fetch+bounce [9].

In WTF, an attacker exploits the behavior of store-to-load shortcut, that is, the CPU will matching a pending write instruction to load instruction. If done in transient domain, the WTF algorithm will speculatively match the write and the load instructions if some of their least significant bits of the addresses match. In the case that the wrong match occured, it will not be visible to the outside world, however it does leave microarchitectural traces (i.e. in cache) [9].

Next building block, the data bounce, exploits the fact that the permission bit is not checked when storing write instruction to store buffer or during store-to-load forward. An attacker leverages this fact to check which virtual addresses are valid (mapped to physical addresses). The attacker chooses an arbitrary virtual address and writes to it. Immediately after writing the write has been enqueued to store buffer, but not yet processed. When the attacker tries to load data from the same address, the store-to-load forward happens if the virtual address is backed by a physical page. To not corrupt data the attack is executed in transient domain. If the mapping is valid, the attacker is able to encode dummy data, initially issued as a write instruction and then forwarded to load instruction, to cache and retrieving it by using FLUSH+RELOAD[18] cache attack [9].

Third building block, fetch+bounce, expands data bounce and the fact that the number of times the attacker has to repeat it to successfully encode dummy data to cache directly corresponds to which address mappings are cached in TLB. One try means the mapping is valid and was already cached in TLB, two tries mean the mapping is valid but it wasn't cached yet on the first try, and three or more tries mean mapping is not valid. This shows that μOPs also leave traces on TLB. FLUSH+RELOAD[18] is used to check if some dummy data was encoded to cache [9].

Fourth building block, speculative fetch+bounce, adds on to the logic of first three by adding Spectre [13] gadgets. Using the spectre gadget allows an attacker to speculatively leak data writes, in this case secrets, from any source, even kernel. Additionally, the secrets are not encoded in CPU cache but TLB instead [9].

Mitigation: Post Coffee Lake R processors include physical mitigations against WTF exploits. However, Intel's concurrent work [6] showed that the mitigations weren't enough. Even using the $mfence$ serializing instruction is not enough to prevent the WTF exploits. Using KAISER [11] mitigation proved itself as successful in some cases, however not in all cases [9].

For successful countermeasure on software level, all store buffer entries should be overwritten for every context switch between kernel and user. Intel updated $verw$ instruction to do exactly that. Operating systems should be enforced to issue a dummy $verw$ instruction on every context switch. However, there is still an attack vector called store-to-leak, not presented in this paper but is analyzed in original Fallout paper, which is assumed to bypass this countermeasure [9].

Lastly, to mitigate these attacks, spectre gadgets should be identified and removed from software [9].

4 ZOMBIELOAD ATTACK

Zombieload is a side-channel transient-execution attack also known as microarchitectural data sampling (MDS) attack. It exploits the way CPU speculatively matches data from the LFB after a cache miss occurs in L1 cache. Additionally, the LFB is shared between all logical CPUs, therefore does not differentiate between processes or privileges. When a load instruction is issued the CPU first checks its L1 cache. If the cache miss occurs, a fill-buffer entry is given to the load instruction. Simultaneously, an entry in the load- and fill-buffer is reserved. The same happens for store instructions - an entry is reserved in the store- and fill-buffer. At the moment the data arrives from the memory, the load instruction can retire and the corresponding load- or store- and fill-buffer entry is freed [16].

When a fault is introduced during memory load, a microcode assist is required. The microcode assist might read old values before detecting them and executing the load again - an attacker can leverage this behaviour to leak secrets. Additionally, it is not possible to control data leaked by specifying an address, e.g. only in-flight values of the LFB present at the moment of the attack can be leaked. However, it is not entirely clear how this happens in the case of Zombieload attack. As data leakage was observed only on Intel CPUs, it indicates that it is due to the implementation issues. Therefore, the authors provide a hypothesis on why and how this leakage works, leaving it to be proven or disproven by future research [16].

Stale-Entry Hypothesis: When a fault triggers the microcode assist, the pipeline is flushed. Instructions being executed by the CPU at the time of an pipeline flush still have to finish execution to not stall the CPU. For that purpose, they are optimistically matched with the values currently stored in the LFB. As every load instruction has an entry in the load buffer as well as the LFB, only part of the physical address is required for an instruction to be matched with the value. As the LFB is being shared among hyperthreads, it can contain their values as well. That might cause the load instruction to be matched with the wrong values [16].

It is suggested that the LFB might not be the only leakage source. Two experiments were performed to confirm that assumption. In the first experiment it was shown, that the LFB indeed is the cause for leakage. The experiment was firstly set up by marking pages as uncacheable and also flushing them from the cache. Subsequently, a secret was written to the page. This setup caused every memory load from that page to ignore caches and go directly to the LFB. Leakage of 5.91 B/s was observed, confirming that the LFB does leak data.

In the second experiment, accesses to the LFB were avoided by using Intel TSX transactions. For the setup itself, a memory location was initialized with some value at first, and then written to inside of the TSX transaction. The cached write was then flushed from the transactions write-set in L1, which resulted in a zombie load. With this setup it was possible to leak at much higher speeds (in order of KB/s) than in the experiment one [16].

In spite of that, the second experiment confirmed that the LFB is not the only leakage source, yet the actual leaking microarchitectural element remains unknown [16].

4.1 Classification

Meltdown and Spectre have shown, that due to the performance optimizations, security risks lie deep inside of CPU architectures. They've caused the CPUs to be looked at from security's perspective,

	RIDL	Zombieload
Leakage Source	Fill Buffer	Fill Buffer
Leaked Data	Uncached Loads and All Stores	All Loads and All Stores
Exploited Fault	Page Fault	Microcode and Page Fault
Fixed with Countermeasures	Yes	No
Works on MDS Resistant CPUs	No	Yes

Table 1: Zombieload vs. RIDL comparison [16].

creating a starting point for new possible attacks and research. Both are possible because of the out-of-order execution of instructions technique which lets the CPU to calculate independent instructions in different order to prevent CPU from waiting. Additionally, the CPU employs speculative execution to guess which instructions to execute out-of-order when a stall, such as fetching data from memory, occurs. Meltdown and Spectre use these optimization techniques to perform instructions they should not. Spectre mainly focuses on exploiting how branch prediction units work, whereas Meltdown is based on transient execution and fault suppression.

Following the principles of speculative execution that Meltdown and Spectre have shown to be an attack opportunity, Zombieload and RIDL[17] attacks have evolved, classified as MDS attacks. These attacks focus on different microarchitectural units, exploiting the way they work to leak secret data. They're called microarchitectural data sampling attacks because they leak samples of data which they have very little control over out of the LFB. They use additional techniques to fit these samples together like puzzles, to interpret the in-flight data passing through those units. While Meltdown and Spectre serve as pillars for these two attacks, RIDL can be seen as a rival to Zombieload. In general, RIDL only uses straight-line code to leak data present in the LFB. It relies only on the speculative execution of instructions when an uncached page is tried to be accessed; in reality, the CPU returns some value from the LFB hoping its the value instructions are requesting, only to later find out it made a wrong decision, therefore reverting the incorrect instructions and repeating them with the correct value. On the one hand, RIDL is based on common principles that Meltdown and Spectre share. On the other hand, Zombieload relies on Meltdown to a greater extent and using it as an underlying leverage in some of its variants (by using fault suppression).

Authors compare Zombieload to traditional memory-based side-channel attacks (from the viewpoint of target control) which is able to break side-channel resistant applications. They follow the state-of-the-art Zombieload classification which is Meltdown-type transient-execution attack, and additionally divide it into 3 sub-classifications based on the chosen variant: Meltdown-US-LFB for variant 1 - exploits supervisor to leak fill buffer data, Meltdown-MCA-TAA - transactional asynchronous abort triggers microcode assist, and Meltdown-MCA-AD - changing accessed and dirty bit activates microcode assist. They also propose the name Meltdown-P-LFB for RIDL [16].

In table 1, an overview of similarities and differences between Zombieload and RIDL is given.

	V 1		V 2		V 3	
Data Leak Flow	Win	Lin	Win	Lin	Win	Lin
1, 2 (Privileged)	N	D	D	D	Y	N
3, 4, 5 (Unprivileged)	Y	Y	D	D	Y	Y

Win - Windows, Lin - Linux, Y - Yes, N - No, D - Depends, V - Variant
Privileged and Unprivileged represent the privilege level of an attacker

Table 2: Overview of different variants to be used for different data leak flows [16].

4.2 Data Leak Flows

To successfully mount the Zombieload attack it is assumed that an attacker can execute unprivileged native code on victim's machine running a trusted operating system and using Intel CPU with hyperthreading enabled. In general, it is only possible to leak data within a current physical CPU core. Following data leak flows are possible [16]:

(1) ***User process to user process:*** attacker can leak data from another user-space application running on the same physical core and different thread (e.g. sandboxed JavaScript code can leak passwords stored in web browsers).
(2) ***Kernel-space to user-space:*** cross privilege leak is possible to load kernel data from user space. Attacker has to execute attack either on the same logical core or sibling logical core.
(3) ***SGX enclave to outside:*** while the code is executing inside of the Intel SGX enclaves, mounting the attack in a different thread on the same physical core leaks otherwise isolated data.
(4) ***Guest VM to guest VM:*** while running in an virtual machine, the attacker can run any code and leak data of another VM. Both have to be located on the same physical cores and in different threads.
(5) ***Host to guest VM:*** attacker mounting the Zombieload inside of a virtual machine guest can leak the data of its host.

4.3 Attack Variants

Underlying building block of the Zombieload attack is a *zombie load* - a microcode assisted transient load of the wrong data. Because these loads pose a fault, they cannot executed completely and need to be re-issued. Authors point out five different variants for Zombieload, however only first 3 have unique features. These three variants are included in Table 2, where their application possibility for different data leak flows is given. All 5 variants are given as follows [16]:

(1) ***Kernel Mapping:*** A setup with two different addresses is required - an user virtual address u and a kernel address k. It is required for both to map to the same physical page. Accessing the u should be a valid way to access the content of a physical page, whereas accessing the kernel address k should introduce a fault and require a microcode assist. To create this setup, firstly, a normal 4 KB page with the user virtual address u is allocated. Secondly, the kernel address k pointing to the same physical location as u is calculated (usually, the kernel pages are 2 MB in size, but the setup works with smaller pages, e.g. 4 KB, as well). This is done on the basis of the k's physical address and the base address of the direct-physical map (u's physical address is retrieved through $/proc/(pid)/pagemap$ or by using a side channel). To trigger a zombie load, a cache line is flushed through u and then accessed through k. Figure 4 depicts the described setup. Variant 1 doesn't work on Meltdown-resistant machines.

(2) ***Intel TSX:*** For this variant some user accessible virtual address u which maps to a physical page p is required. After performing a valid read to it inside of the TSX transaction, it is ensured that the address is cached in L1 cache as a part of the transaction's read-set. Flushing the u from withing a transaction causes it to be flushed from the transaction's cached read-set, thus resulting in a zombie load. This variant works on Meltdown-resistant machines, but requires Intel TSX extension to work.

(3) ***Microcode-Assisted Page-Table Walk:*** This variant needs a setup with 2 different user accessible virtual addresses u_1 and u_2. Both addresses must map to the same physical page p (i.e. shared pages). Address u_2 has to have the accessed bit set to 0 in the page-table entry. By accessing p through u_2 at a later point causes the u_2 page-table entry to have the accessed bit set to 1. As this cannot be done by the page-miss handler, microcode assist is required to repeat the page-table walk to set the accessed bit [10]. When performing the access to u_2 in a transient way, the access bit cannot be set architecturally, thus the zombie load can be achieved for every access. For this setup a privileged attacker in Linux environment is assumed because clearing the accessed bit of v_1 is required. In the case of Windows, the attacker does not need to be privileged due to the fact that the OS clears the accessed bits periodically.

(4) ***SGX Abort Page Semantics:*** whenever a virtual address v mapped to the physical page p reserved as an SGX enclave is tried to be accessed from outside of the enclave, the microcode assist is triggered. When this happens, zombie load from the LFB can be observed during the microcode execution.

(5) ***Uncacheable Memory:*** similar to variant 4, but instead of an SGX enclave page, memory page marked as uncacheable is used. In this variant, the page miss handler issues a microcode assist which results in zombie load.

In general, all variants require having a setup prepared in advance to leak arbitrary data successfully. The common ground to all is to have a cache line with some virtual address mapped to a physical page. After that, the virtual address is flushed using the *clflush* instruction. As this is only the common part for all variants, every variant has to additionally satisfy the variant specific part of the setup as well.

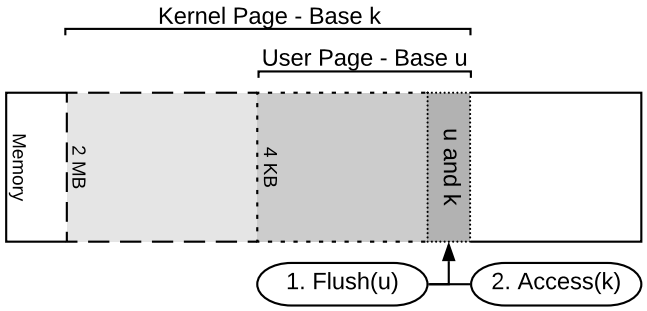

Figure 4: Variant 1 - Kernel Mapping [16].

In the case of variant 1, Meltdown [14] is leveraged to read from the kernel address k. Because Meltdown is used, exception prevention, handling, or suppression mechanisms must be used. For variant 2 and variant 3 there is no need to handle exceptions due to the Intel TSX transactions usage and transient reads respectively [16].

Leaking the LFB Entries and Targeting Their Bytes: Mentioned attack variants are used to leak the in-flight data from the LFB and do not leak any data from addresses or pages mentioned; only a value referenced by the LFB entry of size 64 bytes (which we cannot control) is leaked. Nevertheless, it is possible to target specific bytes within the LFB entry by adjusting the 6 least-significant bits of the flushed virtual address, i.e., the 6 LSB from the virtual address reference the 64 bytes within the LFB. Because only small parts of the whole are leaked arbitrarily, the attacker needs to be synchronized with the victim to effectively filter out the information he intends to leak [16].

4.4 Synchronization and Noise Filtering

Because the Zombieload leaks arbitrary data which cannot be entirely controlled an attacker has to be synchronized with the victim, i.e., the attacker has to mount the Zombieload and start to leak values when the targeted victim's secret data is in-flight. This can be achieved, for example, by observing the cache state for a particular addresses to be cached (i.e. a cache based trigger) or getting the victim to load the secret data into the line-fill buffer. Additionally, noise filtering may be required to leak just the relevant secret data and ignore the irrelevant one [16]. The Zombieload article contains multiple case study examples and noise filtering techniques, but here we omit most of them.

A practical example of the cache based trigger would be observing if a line of shared library's code has been executed and cached afterwards. In the case of OpenSSL, this is the *aesni_set_encrypt_key* line, which loads the encryption key of AES into the memory. A cache is constantly monitored whether it contains a cache line with function's virtual address. At the moment the monitored cache line is detected the Zombieload is mounted. This approach already

filters out unimportant data that was present in the LFB before the AES key was loaded [16].

A practical example of getting the victim to load its secret data into the LFB could be a cloud scenario where the attacker is using a VM located on the same core as the victim. If the victim has SSH enabled and the attacker knows its IP address, he mounts the Zombieload and then repeatedly tries to connect through. Choosing the password doesn't really matter, as this step is only to get the victim to load the original password into the LFB. By observing which values repeat the most throughout many attempts, the attacker could derive the SSH password.

Domino Attack: When leaking value independent bytes which look like a random byte string, e.g. AES key, it is required for the attacker to be able to distinct them from the noise and reconstruct the original secret in the correct sequence. This can be achieved by so called domnino attack shown in figure 5. Because the Zombieload can perform certain amount of computations while still in the transient domain, a domino byte can be constructed. The role of this byte is to connect two adjacent bytes trying to be leaked. It is constructed by taking x number of LSB bits from a byte K_n and concatenating them with $8 - x$ MSB bits of the adjacent byte K_{n+1}. For example, when constructing a (4, 4) domino byte 4 LSB bits and 4 MSB bits are taken from bytes K_n and K_{n+1} respectfully and then concatenated. Domino bytes are leaked together with the secret. By observing their frequencies it is possible to determine the order of the original byte string while filtering out the leaked noise [16].

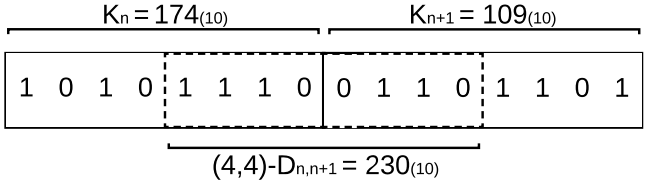

Figure 5: Domino Byte [8].

5 RESULTS

Variants 1, 2 and 3 were tested whether it was possible to mount them on different environments. Results are given in table 3. It was possible to mount variant 2 on every CPU supporting Intel TSX, and variant 1 and 3 on different on microarchitectures without hardware mitigations against Meltdown [14].

Additionally the performance of each variant was evaluated on a Intel i7-8650U processor. While a specific value was read on one logical core, Zombieload leaking data was mounted on the sibling logical core for 10 s. Number of successful and unsuccessful recoveries was recorder. In the case of Variant 1, while using Intel TSX exception suppression 5.3 kB/s average transmission rate with a true positive rate of 85.74% was achieved. For variant 2, an average of 39,66 kB/s and a true positive rate of 99,99% was achieved. For variant 3 an average transmission rate of 7.73 kB/s and a true positive rate of 76.28% while using Intel TSX was achieved [16].

			Variant		
Setup	**CPU**	**μ-arch**	**1**	**2**	**3**
Lab	Core i7-33630QM	Ivy Bridge	Y	D	Y
Lab	Core i7-6700K	Skylake-S	Y	Y	Y
Lab	Core i5-7300U	Kaby Lake	Y	Y	Y
Lab	Core i7-7700	Kaby Lake	Y	Y	Y
Lab	Core i7-8650U	Kaby Lake-R	Y	Y	Y
Lab	Core i7-8565U	Whiskey Lake	N	D	N
Lab	Core i7-8700K	Coffee Lake-S	Y	Y	Y
Lab	Core i9-9900K	Coffee Lake-R	N	Y	N
Lab	Xeon E5-1630 v4	Broadwell-EP	Y	Y	Y
Cloud	Xeon E5-2670	Sandy Bridge-EP	Y	D	Y
Cloud	Xeon Gold 5120	Skylake-SP	Y	Y	Y
Cloud	Xeon Platinum 8175M	Skylake-SP	Y	D	Y
Cloud	Xeon Gold 5218	Cascade Lake-SP	N	Y	N

Y - Yes, N - No, D - Intel TSX is disabled

Table 3: Zombieload Tested Environments [16].

5.1 Case Study Results

Zombieload was implemented in some real world scenarios to show the possibility of its application in practice. In the following subsection, a brief overviews with results for AES-NI key leakage, cross-VM covert channel, and browsing-behavior monitoring case studies are given. The paper additionaly presents SGX sealing key extraction and trageted data leakage case studies 1.

AES-NI Key Leakage: It is shown by the Zombieload that the timing and cache-based side-channel resistant AES-NI [12] key can be leaked. In the setup the OpenSSL cryptographic library and a 128 bit AES key is used for encryption. The attack uses a cache-based trigger, i.e. observing the code line which loads the key into the memory to be cached, acting as an incentive to mount the Zombieload. Additionally, the domino attack is used to reconstruct the order of leaked key bytes and to filter out noise. Data is leaked through many iterations of key loads. Using the variant 1 in a cross-user-space scenario, the attack is evaluated by constantly leaking the data bytes until the bytes with highest probability are the same as the correct key [16].

The final result shows that it's possible to recover the key on average in under 10 seconds throughout approximately 10 000 iterations [16].

SGX Sealing Key Extraction: It was shown by the Zombieload that register values can be observed during an enclave execution. From the register values the 128-bit sealing key can be reconstructed to decrypt the long term EPID private attestation key. A code part, which loads contents stored in an interrupted enclave's saved state into the registers, was identified and repeated multiple times. This allowed register values to be reconstructed at the moment the key used for decrypting the long-term private attestation key was loaded. Therefore, recovering the latter key was possible [16].

Hundred key recovery experiments with random keys were performed. After reconstructing register values to get the key candidates, in 30 out of 100 experiments, key candidates had an average

Website	Minimal	Average	Maximum
nytimes.com	1	1	3
facebook.com	1	2	4
kernel.org	2	6	13
gnupg.org	2	10	34

Table 4: Number of required accesses to recover a website name [16].

entropy of 8.8 bits. In other 70 experiments, on average 63% of key bytes were recovered correctly. The incorrect bytes were usually at the beginning of the key [16].

Cross-VM Covert Channel: In the Cross-VM scenario, an attacker is able to leak data between two virtual machines. For the setup, a sender is constantly loading data into a register (this is done only to speed up the process) and the receiver mounts the Zombieload to leak them. Transient error detection technique is used to filter out any noise not coming from the victim. The technique is to build specially crafted packets in the transient domain, which are able to detect isolate data from the noise and construct the original secret in the correct sequence [16].

The scenario is tested in a lab environment and a public cloud. It is shown, that the error free transmission of data is possible between two virtual machines running in QEMU KVM on an Intel I7-8650U processor in the lab setup, and two VMs running CentOS 7.6.1810 with a kernel version 3.10.0-957 on an Intel Xeon E5-2670 in the cloud as well. For both setups the Variant 1 is used. In the lab setup, leakage speed of 26.8 kbit/s with the usage of Intel TSX for fault suppression is observed. In the cloud setup 1.99 kbit/s leakage speed is observed. In the latter case, TSX was not available [16].

Browsing-Behavior Monitoring: The Zombieload[16] demonstrates the ability of detecting specific byte sequences to fingerprint a web browser session. Two specific attack are presented, namely keyword detection and URL recovery. The former compares compares bytes leaked from the LFB to a predefined list of 4 to 8 byte long keywords and leaking the keyword's list index. The latter is used to recover URLs visited by initially detecting the 'www.' sub-string from the HTTP requests and leaking further bytes correspondingly. Both have the quality of being executed in the transient domain. For both attacks the Variant 2 is used [16].

Attacks were performed on an unmodified Mozilla Firefox 66.0.2. The keyword detection attack reliably matched keywords on the first access of highly accessed websites (e.g. nytimes.com), and with the probability of 60% for static websites (e.g. gnupg.org). For URL recovery attack the number of necessary refreshes was counted until entire URL including top-level domain was recovered. The number was depending whether the website was highly dynamic or static - the results are given in Table 4 [16].

6 COUNTERMEASURES

Zombieload leaks secret in-flight data in the LFB from other processes. Additionally, the Zombieload only leaks data across or within logical cores and can not leak across physical CPU cores. The most obvious mitigation is to disable hyperthreading. But as this would cause significant performance degradation (up to 30%)[5], different approaches should be taken in practice [16].

Co-scheduling technique [15] could be used to prevent different protection domain code from being executed on the same core. By using this strategy, it must be additionally ensured that both sibling cores go into the kernel mode when one tries to access kernel entries [4]. In any case, the authors of Zombieload see co-scheduling as not fully effective mitigation technique [16].

Because Zombieload leaks arbitrary data residing in the LFB, buffers and caches must be flushed on every context switch (the L1 cache and store buffer). Nevertheless, flushing the L1 cache is considered too slow (on average 1070 CPU clock cycles) to be performed on every context switch [16], thus another solution must be found. Intel published update for microcode that allows LFBs, load ports and store buffers to be flushed. This is done with *verw* instruction, which has to be used in combination with speculation barriers such as *lfence* [17]. Nonetheless, after applying this mitigation, it is still possible to leak data with less than 0.1 B/s [16].

A way to disable Intel TSX should be offered by Intel microcode update as this would stop an attacker from using variant 2. For preventing the variant 3, the operating systems should always be setting the dirty or accessed bits in page tables. For variant 4, disabling the Intel SGX would stop an attacker to leak data. Variant 1 can not be mitigated on machines vulnerable to Meltdown. Countermeasure for the variant 5 is not given [16].

In sandbox environments, underlying instructions that are necessary to successfully mount the Zombieload should be disabled (e.g. clflush) [16].

7 CONCLUSION

Microarchitectural data sampling attacks, leaking data from the line-fill buffers are freshly discovered types of attacks which open many new security related questions that the Intel's CPUs should address. It was shown that Zombieload is able to exploit CPUs resistant to Meltdown and MDS attacks. By exploiting the way microarchitectural components of the CPUs are used for performance optimizations, poses a question whether the software mitigations are enough or hardware changes should be implemented instead. While multiple mitigation techniques were proposed, disabling the hyperthreading is currently the easiest and most effective way to stop the attack, however it comes with considerable degradation of CPU's performance.

REFERENCES

[1] 2017. Intel® Enhanced Privacy ID (EPID) Security Technology. https://software.intel.com/en-us/articles/intel-enhanced-privacy-id-epid-security-technology [Online; accessed 9. Dec. 2019].

[2] 2018. Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual [Online; accessed 2. Dec. 2019].

[3] 2018. What every programmer should know about memory, Part 1 [LWN.net]. https://lwn.net/Articles/250967 [Online; accessed 8. Jul. 2020].

[4] 2019. Deep Dive: Intel Analysis of Microarchitectural Data Sampling. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling [Online; accessed 2. Dec. 2019].

[5] 2019. Intel Hyper Threading Performance With A Core i7 On Ubuntu 18.04 LTS - Phoronix. https://www.phoronix.com/scan.php?page=article&item=intel-ht-2018&num=1 [Online; accessed 8. Dec. 2019].

[6] 2020. Deep Dive: Intel Analysis of Microarchitectural Data Sampling. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling [Online; accessed 14. Jul. 2020].

[7] 2020. Memory part 2: CPU caches [LWN.net]. https://lwn.net/Articles/252125 [Online; accessed 8. Jul. 2020].

[8] Daniel P. Bovet. 2005. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media.

[9] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[10] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.

[11] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems*, Eric Bodden, Mathias Payer, and Elias Athanasopoulos (Eds.). Springer International Publishing, Cham.

[12] S. Gueron. 2012. Intel Advanced Encryption Standard (Intel AES) Instructions Set - Rev 3.01.

[13] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

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

[15] J.K. Ousterhout. 1982. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*. 22–30.

[16] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*.

[17] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.

[18] Yuval Yarom and Katrina E. Falkner. 2013. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*.