UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Using ZombieLoad for Co-location Detection between microVM

Co-location Erkennung zwischen MicroVMs mittels ZombieLoad

Bachelorarbeit

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Thilo Beccard

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

Lübeck, den 20.01.2023

Abstract

The goal of this bachelor thesis was to achieve co-location detection between two firecracker instances by using the transient execution attack called ZombieLoad. ZombieLoad uses leakage from the line-fill-buffer between the L1 and L2 cache to recover single bytes. Multiple threat models were examined in order to measure the effects of isolation through firecracker as well as background noise on the attack's success. In a second experiment, ZombieLoad was used to recover byte strings from the attacked process, providing necessary information for identification. The comparison of different execution environments showed that isolation provided trough the use of firecracker micro-VMs has a negligible influence on the performance of the ZombieLoad attack. Further experiments regarding the identification of a threshold of leaked bytes and the estimation of the total attack runtime revealed that detection of co-location is a realistically feasible use case for a ZombieLoad attack.

Abstract

Das Ziel dieser Bachelorarbeit war es, eine *Co-Location-Detection* zwischen zwei Firecracker-Instanzen zu erreichen, indem eine *transient execution attack* namens ZombieLoad verwendet wurde. ZombieLoad nutzt Leaks aus dem *Line-Fill-Buffer* zwischen dem L1- und L2-Cache, um einzelne Bytes wiederherzustellen. Es wurden mehrere Bedrohungsmodelle untersucht, um die Auswirkungen der Isolierung durch Firecracker, sowie die des Hintergrundrauschens auf den Erfolg des Angriffs zu messen. In einem zweiten Experiment wurde ZombieLoad verwendet, um Byte-Strings aus dem angegriffenen Prozess wiederherzustellen, die die notwendigen Informationen zur Identifizierung liefern. Der Vergleich verschiedener Ausführungsumgebungen zeigte, dass die Isolierung durch die Verwendung von Firecracker-Micro-VMs einen vernachlässigbaren Einfluss auf den Erfolg des ZombieLoad-Angriffs hat. Weitere Experimente zur Identifizierung eines Schwellwerts für erkannte Bytes und zur Abschätzung der Gesamtlaufzeit des Angriffs ergaben, dass die Erkennung von Co-Location ein realistisch durchführbarer Anwendungsfall für einen ZombieLoad-Angriff ist.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

_____

Lübeck, 20.01.2023

Contents

1 Introduction

As digitization continues and the resulting need for flexibly scalable server resources for small services increases, serverless computing is becoming an increasingly important concept with expanding possibilities. On the downside, new potential attack scenarios open up and can be exploited.
Firecracker version 1.0 has only been released at the end of January 2022 [fir22a] and has already been used and tested for popular cloud computing services like AWS Lambda for some years [ABI+20]. It might be a new important part of infrastructure for serverless computing. Firecracker is a Virtual Machine Monitor for minimized virtual machines (VMs), each hosting a single service function. The amount of services that Firecracker is capable to run on a single host machine makes it attractive as a possible target for co-location detection based on side-channels between virtual machines, since it is possible for an attacker to be co-located with many different processes. One way to create such a side channel is the ZombieLoad attack [SLM+19], which can randomly leak bytes from the line fill buffer.

In this work, the chosen scenario is a co-location detection [IGES16] based on the ZombieLoad attack, with the goal of detecting and possibly identifying services that are hosted simultaneously on the same Hardware.
At the beginning of this thesis, the fundamentals of Firecracker, ZombieLoad and co-location detection are explained.
Subsequently the original ZombieLoad implementation and optimizations added for this work in order to increase performance and reduce wrong leakage are discussed. In particular, the overall setup, the implementation of victim and attacker are described and evaluated.
In the following chapters, the implementation is used in different attack settings to measure and evaluate the impact of environmental factors and different attack scenarios. The first experiment conducted aims at evaluating the effects of the chosen execution environment and the addition of background noise on the success of the ZombieLoad attack. Runtime and correct byte leakage are compared for each execution environment with and withoud added noise. The goal of the second experiment is to determine the lowest possible threshold of bytes recovered per byte position at which the correct leakage byte can still be chosen with high certainty. Using such a determined threshold, runtime estima-

tions are made for setups with multiple virtual cores in order to deduce the feasibility of the presented attack in a real-world scenario. This work shows that ZombieLoad is a viable attack for use in a firecracker environment, as the impact on the attack due to the isolation provided by firecracker is negligible. Also, it can be demonstrated that ZombieLoad is feasible for co-location detection. Measurements of execution time are used to provide an estimate of performance in an actual attack scenario.
Finally, a conclusion on the outcome is drawn, limitations of this work are outlined and an outlook on future work is given.

2 Background

In this chapter the reader will be introduced to the main concepts used in this work. This includes the VM monitor Firecracker, the ZombieLoad attack, as well as co-location detection.

2.1 Firecracker

Firecracker [ABI+20] is an open source virtual machine monitor developed and used by Amazon Web Services. It is designed for efficient serverless computing. The concepts of serverless computing and Kernel-based Virtual Machine (KVM), which Firecracker is based on, as well as Firecracker itself will be explained in the following.

2.1.1 Serverless computing

Serverless computing gets described [BCC+17] by the amount of control a developer has about the infrastructure the software is deployed on. It gets described on a scale from Infrastructure-as-a-Service, where nearly full control over the hosting system is given, to software-as-a-Service, where a developer has no control about the deployment. This is visualized in figure 2.1. On this scale, serverless computing is a compromise between both extremes. The developer has full control over the code while not having to deal with the operational parts of a deployment. This model of deployment is designed for scalable services and therefore should be running stateless functions that can be scaled and are fail resistant. A serverless platform provides the capability to process events like forwarding web requests to the right process and delivering the response, start and stop instances if needed or manage the logs.

Figure 2.1: Firecracker classified on a scale between Software-as-a-Service and Infrastructure-as-a-Service

2.1.2 KVM

The Kernel-based Virtual Machine (KVM) [KKL+07] is a Linux subsystem designed to use virtualization extensions of hardware [UNR+05] to create and run multiple virtual machines. The hardware extensions are the guest operating mode, a hardware state switch and the exit reason reporting. The guest operating mode grants regular privilege levels where registers and instructions can be selected to be trapped. The hardware state switch enables to switch in and out of guest mode by switching instruction pointer, segment and control registers. The exit reason reporting passes on the cause for a switch back to the hostsystem to enable specific handling of the situation. For a guest system the guest operating mode provides a from the hosting userspace separated memory, but no separated CPU schedule.
To allow KVM virtualization to use these hardware extensions, guest mode has been added to kernel mode and user mode already present in Linux. KVM provides features for running a virtual CPU like reading and writing of registers or interrupt injection, and for allocating and mapping memory to a VM.

2.1.3 Firecracker

Firecracker[ABI+20] is based on the Linux Kernel-based Virtual Machine. The so called MicroVMs running in Firecracker are supposed to provide a fast and memory efficient environment for serverless computing. MicroVMs differ from regular VMs that they are intended to provide equivalent security through isolation while being reduced to the minimal functions. Because of the designed minimalism Firecracker is able to run up to thousands of MicroVMs per server. Each MicroVM is designed to serve only one function, and if needed more MicroVMs with the same function are added. Firecracker differs from common container usage, which are relying on isolation provided through the Kernel, by not only relying on a single Kernel and the resulting compromise of compatibility and security. Firecracker is designed to get rid of the compromise and provide maximal compatibility and security. It differs in the implementation of similar projects by only using KVM as the base and not the combination of QEMU and KVM. Instead of QEMU it uses a newly developed Virtual Machine Monitor with dedicated device model and API. Security is provided trough isolation in individual guest systems, as each function has its own MicroVM. Against micro-architectural side-channel attacks guidance for best-practices is provided[fir22b]. Those include disabling Symmetric-MultiThreading, enabling the Linux kernel mitigations, Kernel Page-Table Isolation, Speculative Store Bypass mitigations, disabling swap and samepage merging and using memory supporting Rowhammer mitigations.

2.2 ZombieLoad

This section explains the ZombieLoad attack and its basic principles, as well as co-location detection.

2.2.1 MDS

Microarchitectural data sampling attacks (MDS attacks) [SLM+19] are described as a type of attack between memory-based side-channels, that use correlate addresses inside the targeted process and transient-execution attacks, which leaks data from a specific addresses. These types of attacks are based on the deficient knowledge of microarchitectural buffers by exploiting microarchitectural faults. This makes use of the property that faults in the microarchitecture are not communicated to the superordinate one. MDS attacks[Shi21] are started with a instruction causing a microcode assists or faults. The transient execution of following instructions is used to gain access to data through an available covert channel. MDS differs from Meltdown-type attacks [LSG+18] in that the leaked address can not be specified and therefor leakage of data regardless from significance.

2.2.2 Line Fill Buffer

The line fill buffer (LFB) is a buffer between the L1 and L2 cache. Every miss in the L1 cache leads to allocation of a LFB entry. This marks for every other load, that the process of retrieving the data is already in progress and is supposed to lower the costs of another miss in the L1 cache.

2.2.3 TSX

Intels Transactional Synchronization Extensions (TSX) [Int22] is supposed to enable the processor to dynamically decide if a thread needs to be serialized and perform this seralization only for these crucial parts. A programmer is also able to specify these parts by them self with so called lock elision. A locked variable is read only. When a transaction is successful executed the memory operations will made be visible for other processes. An unsuccessful execution leads to a so called transactional abort, which means a roll back is performed and all changes are discarded. ZombieLoad uses the TSX feature xbegin, which starts a transnational region an returns a status code. To commit the executed transaction xend is called.

2.2.4 Flush and Reload

Flush+Reload [YF14] is a technique to recover cache lines based on the measured access time. The attack is executed in three steps. First the attacked memory line gets flushed. Then the attacking process waits for the targeted process to access the memory. In the last step the time for reloading the memory is measured because the access time differs depending on the cache level. If the accessed line is already loaded into the L1-cache the shortest time will be measured. The time increases with every rising level the line has to be loaded from. According to the researchers the attack has an above 90% success rate. ZombieLoad uses this technique to recover the leaked bytes.

2.2.5 ZombieLoad

ZombieLoad [SLM+19] is a fault-driven transient-execution attack that still works on Intel CPUs with security measures against microarchitectural data sampling (MDS) attacks.

Functionality

ZombieLoad accesses data from the line fill buffer which is used for load and stores. However, it is unclear whether this is the only point of leakage used by the attack. ZombieLoad uses loads that require microcode assist and are triggered by a fault. During a microcode assist a transient execution window is opened within which a stale value can be accessed for calculations before being forwarded to the correct value. This enables to read and encode stale values from the line fill buffer which can be recovered later. For Recovery the flush and reload method is used. ZombieLoad is not able to leak specific data and leaks current content from the line fill buffer. This attack is able to leak from all privileges and is referred to as a data sampling attack.

Variants

There are three variants of the attack[Ins20]. The first one is "Kernel Mapping" and only works for privileged attackers on machines that are not protected against Meltdown attacks. A huge page with read access through an user accessible address is used. By accessing these page via its kernel address a a microcode assist is created, leading to a Zombieload. The second variant "Intel TSX" uses a user accessible page and purposely placed conflicts in the TSX transaction to archive a transient fault leading to a ZombieLoad. The TSX function xbegin is used to create a microcode assist leading to a falsely executed if-statement. The third variant "Microcode-Assisted Page-Table Walk" used a page with two accessible virtual addresses. One virtual address is used to access the content. When the

accessed bit it cleared and the page is accessed via the second virtual address, the bit has to be set again. This process needs a microcode assist leading to a ZombieLoad and therefore resulting in a leakage.

Similar Attacks

ZombieLoad differs from other MDS attacks like RIDL [vSMÖ+19] and Fallout[CGG+19] especially though the point that it still works on newer Intel CPUs. RIDL leaks from the line fill buffer like ZombieLoad, but only for loads that are not at the time loaded in L1 cache. The attack uses a load to access in-flight data speculatively used by the processor and recovers like ZombieLoad the data via flush and reload. Fallout, on the other hand, leaks from the store buffer. For RIDL and Fallout, the cause of the leakage is also known, unlike the ZombieLoad leakage.

Domino Bytes

The authors of the ZombieLoad paper[SLM+19] suggest a error detecting technique named Domino attack for filtering noise in the leakage. The Domino attack uses domino bytes made out of the high and low nibbles (half a byte) of two successive bytes. This enables to leak the byte from positions shifted only 4 bits to the predecessor. The overlap between the resulting bytes is used to filter leaked bytes for those of which the low nibbles are corresponding to the high nibble of the predecessor as shown in figure 2.2.

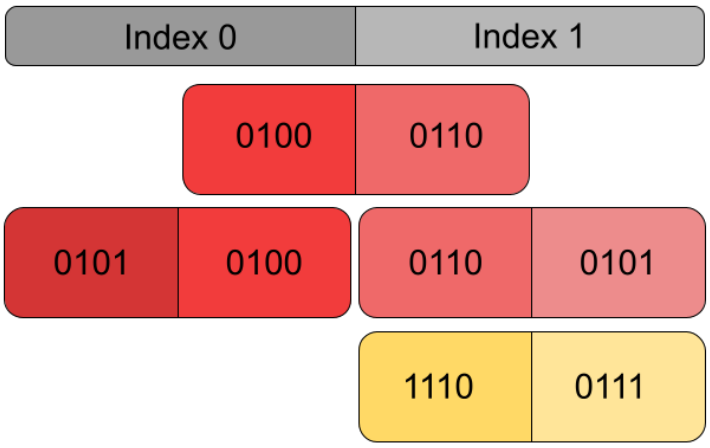

Figure 2.2: Bytes separated into low and high bits. The domino byte connects matching bytes from both indices (red) and helps to exclude the wrong one (yellow).

2.3 Co-location detection

Co-location detection for VMs [IGES16] describes the attempt to gain knowledge about the presence or type of VMs hosted on the same hardware. Other work about co-location also refers to is as "co-residence" or "co-tenancy" [PPL15]. Two services are co-located when they share a host server simultaneously. Co-location detection was previously achieved through different types of attacks, all of which have in common that they use measurable contention about shared resources to create a channel for information leakage. This is possible in a microarchitectural level such as "Prime and Probe" were memory access time is used to detect LLC usage, or at higher levels of shared resources, like with "Network Probing" [RTSS09] based on different network properties. Prime and probe techniques usually create an artificial state in the step referred to as prime to ensure the attack requirements are fulfilled. In the probe step the prepared stated has to be measured in some way, like detecting preloaded values in cache or network traffic. In order to detect a specific target, identifying information is required. In this work Co-location in form of two processes running on the same physical CPU core, and therefore a share line fill buffer, is being studied. The detection is also performed on a microarchitectural level.

3 Implementation

In this chapter, the implementation setup and considerations will be outlined. In addition to the hard- and software used for the general attack setup, implementation optimizations for increasing performance and decreasing wrong leakage are proposed. The evaluation shows that reducing the size of the recovery alphabet can significantly increase the match rate. Moreover, defining a suitable threshold of recovered bytes per position allows us to reduce wrong leakage.

3.1 Attack Execution Environment

Every test was executed on an Intel-Xeon-Silver 4114 powered server with TSX enabled and running Ubuntu 20.04. The used Firecracker version was 1.0. The guest used in the micro-VM was minimized Alpine-OS with gcc installed as all the code is programmed in C.
The attack aims at recovering data from a line fill buffer. In order to simulate a repeated load action in the L1-cache we use the victim implementation provided by Borrello[pie19]. It fills each cacheline of a memory page with the same string and accesses the different lines of the page in a loop to ensure the string is constantly loaded into the line fill buffer. The loop through the lines ensures that each line is already evicted from the L1-cache on access and has to be reloaded to the cache.

3.2 Attacker

Based on the ZombieLoad attack example for Linux userspace, provided with the Paper [Ins20] and an example for domino bytes [pie19] a version capable of leaking domino-bytes was created. In the following the setup of the original implementation of ZombieLoad as well as the idea behind domino bytes will be explained in order to derive the resulting ZombieLoad attack with a 4-bit shift.

3.2.1 Original Implementation

The original implementation starts the attack preparation by checking for Intel-TSX support and notifying if it is not available. This is due to the microcode assist which is used in this version of the attack being triggered by a fault in an Intel-TSX transaction. Next, a

whole memory page and a mapping to the fill-buffer get initialized with null bytes (0x0) and the page gets flushed. A flush and reload threshold is determined by measuring the average reload-time (R) and the average flush-and-reload-time (F) and then calculating

$$\frac{(F + 2 \cdot R)}{3}$$

.
Now the execution of the attack begins. Each execution first flushes the mapping. Then the transaction is started by using successful execution of a TSX xbegin instruction as a condition in an if-statement. The code in the body of the if-statement executes the memory access, leading to the leakage from the LFB followed by a call for xend to end the transaction. As a last step, the recovery of the leaked byte via flush and reload is performed.

3.2.2 Shifted Bytes

Shifting the position of leaked bytes is possible[pie19] by adding bit shifts for the high and low part of the domino byte 2.2.5. The address of the predecessing byte is shifted four bits to the left added with a logical or to the predecessing byte shifted twelve bits to the right and the result masked of with an all high byte (0xff)

3.2.3 Derived Optimizations

With the basic Implementation it would already be possible to leak all bytes and corresponding domino bytes to reconstruct a string. Unfortunately, the data is disturbed by a high hit rate of the first byte on all the following byte positions in the cacheline and the detection of the leakage rate is low with 255 possible combinations. In this section some optimizations of the basic attack implementation with regard to performance and wrong leakage reduction are proposed.

Performance

In order to increase performance, one option would be to decrease the number of possible values of the leaked byte. This set of bytes is referred to as leakage alphabet or recovery alphabet. If only a subset of the possible values for a byte is being used as the leakage alphabet, the recovery can be executed with less flush and reload attempts and a higher match rate. For the first byte of a string this is possible if limiting properties are known, e.g. if a specific character is searched. By only shifting every following byte by four bits from its predecessor, the four high bits are known. Instead of the previously used recovery

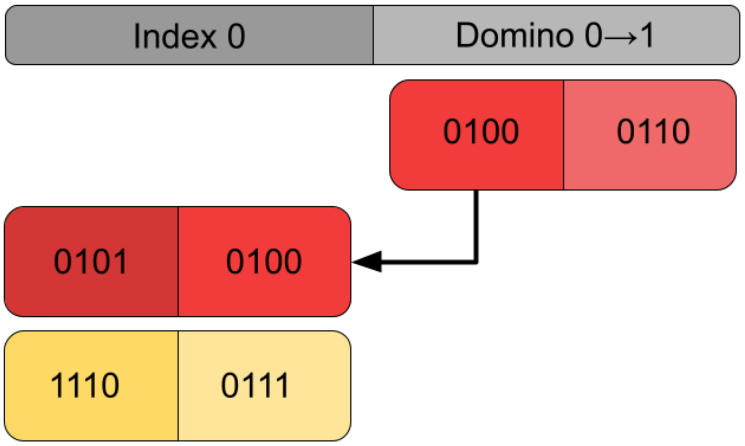

Figure 3.1: The correct of the two most leaked bytes [red] is retroactively determined by the domino byte.

alphabet of 255 values, a smaller recovery alphabet can be used since four known high bits are resulting in only $2^4 = 16$ possible combinations of the lower bits.

Wrong Leakage Reduction

If the first byte is part of the set of possible bytes, it is always also leaked on the following positions of unshifted bytes and with a higher match rate. To mitigate this, a high enough threshold for leaks has to be chosen to ensure that the correct byte is among the to most leaked bytes. With a higher threshold the second most leaked stands out more from the other values. The most leaked following domino-byte then can be used to determine the correct predecessor as shown in figure 3.1.

How Optimization Was Measured

To measure the effect of the optimizations the hits over a time span of one minute where recorded with the test string "load". This test was performed on the first two bytes and their following shifted byte. Now it it possible to calculate the hits per minute, the percentage of correct hits and correct hits per minute.

3.3 Evaluation of Implementation

In order to evaluate the implementation setup, tests were conducted with regard to the speed and the accuracy. The first test for the evaluation of the implementation is the comparison between the basic implementation with full alphabet (bi_fa), basic implementation with a reduced alphabet from the ASCII values "A" to "z" (bi_ra) and the automatic calculated reduced alphabet through domino bytes (dom_alph), also starting with the "A" to "z" alphabet. The hit threshold per position was set high, to five-hundred, and the seven symbols long string "ImpTest" was used. Because of the large alphabet bi_fa did not finish. bi_ra was executed fast, in under thirty-seven seconds, but with the first position leaked correctly. The remaining positions have been mostly affected with wrong leakage of the first byte, except for one single correct hit. dom_alph finished in 1619 second and only 0.26% wrong leakage which, as can be seen in figure 3.2, is a visible improvement. The 4376% longer execution time results from double the amount of byte positions because of the used Domino-Bytes and the high hit threshold per position. Filtering through the reduced alphabet also results in fewer bytes being leaked in the same amount of time.
The unfinished run with the full alphabet shows the need for the reduced alphabet, this has been confirmed by the significant rise in hits with the usage of the reduced alphabet. The amount of wrong leakage at the second position confirms the need for filtering through the used Domino-Bytes and the dynamically adapting alphabet. The wrong leakage when the first byte is included in the leakage alphabet shows that the subsequent filtering between the first and the second most leaked byte by using the Domino-byte is useful.

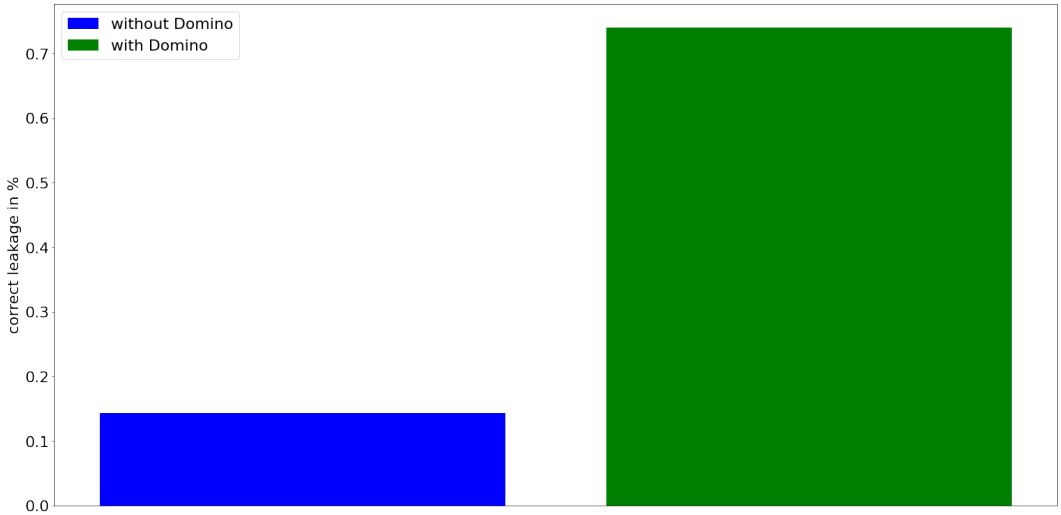

Figure 3.2: Comparison between attack with and without usage of Domino-bytes

4 Experiment: Evaluation of the Execution Environment

In this section the Performance differences of the attack in the different configurations of attacker and victim running on the host-system and inside micro-VM are compared. This experiment is necessary to evaluate the effects of the Firecracker environment and background noise on the ZombieLoad attack. A reliably detectable recovery of correct values is needed to perform the data transfer used in Co-location detection.

4.1 Experimental Setup

The correct leakage and runtime per byte position is measured for a leakage threshold of five-hundred. The leakage threshold is the number of bytes that are recovered per position. The total time is calculated as the sum of the runtime per byte position, until the threshold is reached. The string used for testing was "Exp1". The correct leakage was calculated as the average of correct byte ratio at all four byte positions. An starting alphabet from "A" to "z" is used and the test is repeated for the first two positions of the string to be leaked and the corresponding domino bytes. The experiment was executed for every scenario of attacker and victim being hosted in- and outside of the micro-VMs. Therefore, the four scenarios are:

1. attacker and victim both run on the host system
2. the attacker runs on the host system and the victim runs inside a micro-VM
3. the attacker runs inside a micro-VM and the victim is executed on the host
4. attacker and victim each run in their own micro-VM

To simulate noise of other processes running on the host system, experiments for each of the four scenarios were repeated with additional application of the tool Stress-ng [1].

4.2 Experimental Results and Evaluation

In the following section the results from the different threat models with and without added stress are evaluated.

[1] `https://launchpad.net/ubuntu/+source/stress-ng/0.09.25-1ubuntu9`

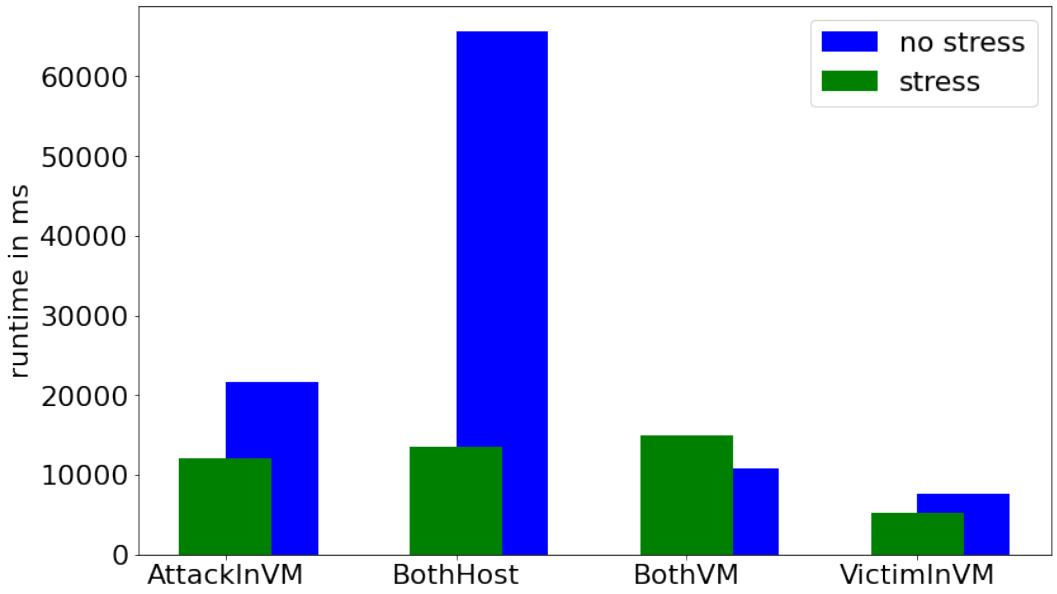

Figure 4.1: Comparison of execution time for each setup.

Simple setup (no stress)

With the attacker and victim both being directly hosted on the host system, the attack worked slowest with an execution time of 65591ms as clearly visible in figure 4.1. On the other hand, the best percentage of wrong leakage was achieved with 0%. The high amount of correctly leaked bytes is due to no other process running on the same CPU core and therefore no load instructions for data to the L1 cache that can be leaked.

Moving only the attacker to the micro-VM had an effect of significantly decreasing the execution time to 21602ms and and increasing the wrong leakage to 5.6%. Moving the victim to the micro-VM decreased the time even further to 7618ms, nonetheless, the wrong leakage observed remained at less than 1%.

Executing both participants in separate micro-VMs achieves a wrong leakage percentage of 2.25% and an execution time of 10757ms.

While the correct byte leakage is fairly high in every scenario, significant differences in execution time can be observed when running the attacker and victim on the host system compared to running one or both of them inside a micro-VM. This can be explained with the noise produced by the individual micro-VMs. This observation will be supported by the findings from the experiments with additional stress which are explained hereinafter.

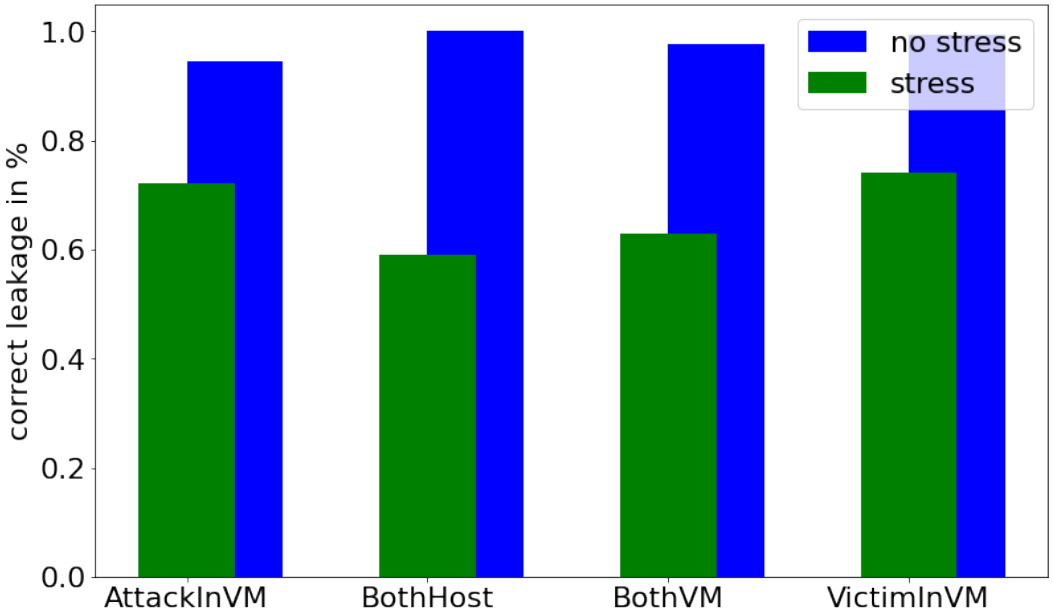

Figure 4.2: Comparison of correct byte leakage for each setup.

Setup with added stress

With stress added, the execution time improved for every scenario compared to the same experiment executed without stress, except when attacker and victim are run in micro-VMs: Compared to the execution time 10757ms in the simple setup, the execution of the attack scenario with additional stress took almost 1.5 times as long (15018ms). Especially the strong time decrease when both actors are run on the host system and Stress-ng is added shows, that a certain amount of background noise is likely to improve the speed. For the previously mentioned scenario in which both are run inside of micro-VMs, the noise provided through the VMs is already suffictient and the additional noise introduced by Stress-ng has a negative effect. A possible explanation for this observation is the background noise affecting the amount of loads to the L1 cache and therefore increasing usage of the line-fill-buffer and its leakage rate. The negative effect of to much noise could be a result from to many load instructions, that are not evicting the targeted data of the victim but only the loaded data from the noise itself.
Adding the tool Stress-ng to each scenario increased the wrong leakage by around 30.9% on average as displayed in Figure 4.2. This had to be expected since other data than the targeted string is loaded to the cache. This is no problem as long as the most leaked byte in the set of the reduced alphabet is still the correct one, targeted in the attack.
The measurements have not shown any sign for negative effects on the ZombieLoad attack cause by the isolation provided through Firecracker micro-VMs. The loss of correct

leakage through a certain amount of noise can be an desirable trade-off for the improvement of the runtime. Therefore the experiment has shown, that not only the data transfer between micro-VMs, needed for co-location detection, is possible despite the background noise, it could even be performed in a shorter runtime because of the background noise.

5 Experiment: Threshold Identification for Recovered Bytes and Runtime Estimation

In this experiment, an attack scenario is simulated with the aim of recovering a string usable for identification of the victim. The goal of the experiment is to find a threshold at which it is apparent whether the victim is highly likely to be identifiable or whether the micro-VM must be restarted to attack another CPU core. The execution time for the determined threshold is used to calculate the expected value of attempts required and the time to perform co-location detection for a process on the hardware used.

5.1 Experimental Setup

For this experiment, attacker and victim each are executed in their own micro-VM and background noise is provided through the tool stress-ng. In the chosen scenario, the attacker has full control over the victim and is able to execute own code in the attacked micro-VM. Therefore, the attacker is able to specifically search for an already known string in the leaked data. In a real threat model this only would be possible if the targeted victim is also owned by the attacker. It is more likely that an attacker would force a target to reload something constantly into the L1 cache by, for example, frequently sending the same web requests. For the sake of simplicity, in this experiment the string known by the attacker is loaded into the L1 cache multiple times in order to mimic the real-world threat.

Since the string searched for is known, it is possible to use the reduced alphabet starting at the first index. For this scenario the same victim as before can be used. The used string is "ExpTwo" and the first three bytes with corresponding Domino-bytes are leaked. The threshold T describes the number of leaked bytes from the alphabet to be recovered. It is decreased from $T = 500$ in steps of fifty until $T = 150$, at which the string cannot be recovered anymore. The execution time and correctness of recovered bytes are also measured. The reboot time for a micro-VM was estimated at five seconds.

After identifying a suitable threshold, the chosen threshold is used to estimate the runtime for the execution of a threat model and to consequently evaluate the feasibility of the attack in different scenarios.

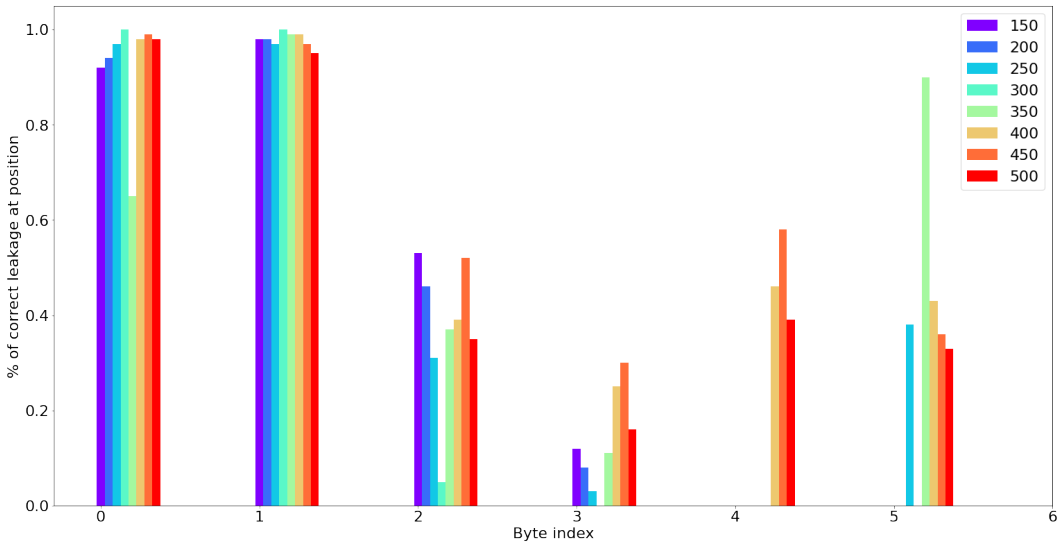

Figure 5.1: Comparison of correct byte leakage for different thresholds at each position. Each color represents a threshold.

5.2 Experimental Results and Evaluation

In the following section, the results from the comparison of different thresholds T are described and evaluated. Furthermore, the total attack runtime is estimated based on a chosen threshold of $T = 450$ and the measured attack duration from the experiment.

Threshold Identification

For every threshold $T \geq 400$ the correct bytes have been leaked. By comparing the leakage count of the first and second most leaked bytes, it is possible to draw a conclusion regarding the certainty of correctness of the first most leaked byte. It is possible to see that the most leaked byte for the threshold of $T = 500$ is at its best position being leaked 97.6 times more than the second most leaked byte. A its worst position, the byte is still leaked 1.39 times more. This is nonetheless enough to detect it as the correct byte, which is validated through its preceding and following bytes, that got at least 1.7 times more leaks than the second most leaked byte and therefore imply that with high certainty the correct sequence of bytes has been leaked.
At a threshold of $T = 350$ the attack fails for the fist time at the second Domino-byte: The correct byte is only being leaked 34% as often as the most leaked one and is therefor only the third most leaked byte for this position, resulting in a continuation of the error to the next position. For the following estimation, a threshold of $T = 450$ leaked bytes is chosen, as it allows the discovery of correct leakage bytes with a high confidence as shown in 5.1.

Runtime Estimation

To estimate the runtime for the execution of a threat model the threshold is set to $T = 450$ as this is high enough for the leaked bytes to not be affected too much by fluctuations in the measurements of the correct leakage. On this setup the execution of the attack took about twenty-seven seconds. With the additional reboot time of a micro-VM, one attack execution would need thirty-two seconds. The used setup uses a CPU with forty virtual cores. The assumption is made, that for every core, the chance for executing a given micro-VM is uniformly distributed.
The probability for launching the attack on the right core at least once for a given number of attempts n can be calculated as follows:

$$n \geq \frac{ln(1-\alpha)}{ln(1-\beta)}$$

with α being the lower bound of probabilities that needs to be achieved and β being the likelihood for launching the attack on the right core.
In order to achieve high confidence of having performed the attack on the correct core at least once, with $\alpha = 0.9999$ the lowest required probability is set to 99.99%. With the probability of launching the attack on the correct one out of c cores

$$p = 1 - \frac{1}{c}$$

β is set to $1 - \frac{1}{40} = 39 \cdot 40^{-1}$.

Now, the number of attempts can be calculated as follows:

$$\frac{\ln 10^{-4}}{\ln 39 \cdot 40^{-1}} \approx 364$$

The 364 attempts result in an total runtime of $364 \cdot 32s = 11648s$. This total attack execution time of 3.2 hours is still shorter than the maximum lifetime that is probably configured for a micro-VM, the attack execution would therefore be feasible in this scenario. In order to be executed on a larger scale, more attacking microVMs will be needed, as the estimated runtime with a second server already would be more than 2.6 times as long and for three server would increase to over four times the calculated execution time.

This estimate shows that the co-location detection is feasible for a small amount of possible cores, but for large data centers for which firecracker is designed, the complexity and runtime quickly exceed the maximum microVM lifetime of one day.

In a real working environment, this attack would probably only be feasible at the tested speed if the targeted process is designed to be easily detected. If a regular program is the target, a much higher threshold will be required until a byte is statistically distinctly visible. This results in an also significantly higher runtime.
Furthermore, an known identifier such as the given search string in the previous experiment is required. Since Firecracker is designed to scale dynamically for services, which for instance could handle web requests, a constant part of a request header might be feasible to use as such an identifier.

6 Conclusions

In this chapter, the contents of this thesis will be summarized and discussed. Limitations of the experiments and questions arising from them are outlined.

6.1 Summary

The aim of this work was to examine the impact of Firecracker micro-VMs and an emulated environment onto the ZombieLoad attack and to determine whether it is a feasible attack to perform co-location detection. In order to do so, an optimized implementation was suggested to leak all bytes and the corresponding domino bytes. Two main experiments were executed:

1. Multiple threat models using different combinations of host system and micro-VMs to run the attacker and the victim were compared in terms of the wrong leakage ration and execution times. Moreover, all scenarios were evaluated with the additional use of the tool Sress-ng for creating background noise.

2. Different thresholds for recovered bytes were tested in order to determine the lowest threshold at which it is apparent whether the attacker will likely be successful or whether the attack has to be relaunched to another CPU core. From the identified threshold, estimations were made with regard to the total runtime of an attack in setups with multiple cores on which the target might be running.

In the first experiment, the effects of Firecracker and background noise have been measured and evaluated. The results show that the isolation through micro-VMs has a negligible effect on the correct byte leakage and the generated background noise can even increase the attack performance despite the increase of wrong leakage. The second experiment reveals that by determining the optimal leakage threshold, an estimate of the time and attempts required to perform a co-location detection, on that specific hardware, can been calculated. The runtimes estimated indicate that the examined attack present a realistic threat for the Firecracker environment in case that few cores are being used, nonetheless the attack runtime rises rapidly with a scaling environment with increasing numbers of cores.

6.2 Discussion and Future Work

In the experiment chapter 4, background noise is added using the tool Stress-ng. Naturally, the background noise in real processes is less random and closely related to the process running. Therefore, it would be interesting to evaluate the influence of background noises from different types of applications on the ZombieLoad attack and to examine the ability to identify the running process from the type of background noise produced.

The estimated runtime of the co-location detection assumes a uniquely identifiable byte string at this point, but especially when identifiers like request headers are used, the chance of overlapping parts of byte strings increases. This is problematic for the execution of the attack, because a chosen identifier also used in multiple different processes could trigger a false positive for a detection. This raises the question whether more unique substrings that can serve as identifiers in more realistic scenarios could be determined. Furthermore, it would be interesting to see if avoiding common prefixes (e.g. standard headers in a web request) and therefore starting the detection not at byte 0 but at a byte with a higher expected entropy.

The calculated runtime estimate has shown that with multiple CPUs or even a whole data center the expected time needed for a successful co-location detection increases rapidly. Therefore the use of multiple attacking processes should be considered. Further research is required on the topic of orchestrating multiple attacking processes for co-location detection.

Also, the experiments have been performed under controlled conditions, a real working environment and especially processes loading similar data repeatedly could probably effect the percentage of wrong leakage negatively or in a unforeseen way. Therefore, experiments on an actual running Firecracker environment would yield interesting insights on the practicability of the attack.

References

[ABI+20] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 419–434. USENIX Association, 2020.

[BCC+17] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. *Serverless Computing: Current Trends and Open Problems*, pages 1–20. Springer Singapore, Singapore, 2017.

[CGG+19] 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. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[fir22a] firecracker-microvm. Firecracker. `https://github.com/firecracker-microvm/firecracker/releases/tag/v1.1.3`, 2022.

[fir22b] firecracker-microvm. Firecracker. `https://github.com/firecracker-microvm/firecracker/blob/79ed3c4dd1ad2d2db632cbec7341de47d081437b/docs/prod-host-setup.md#mitigating-side-channel-issues`, 2022.

[IGES16] Mehmet Sinan Inci, Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. Co-location detection on the cloud. In François-Xavier Standaert and Elisabeth Oswald, editors, *Constructive Side-Channel Analysis and Secure Design - 7th International Workshop, COSADE 2016, Graz, Austria, April 14-15, 2016, Revised Selected Papers*, volume 9689 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2016.

[Ins20] Institute of Applied Information Processing and Communications (IAIK). Zombieload. `https://github.com/IAIK/ZombieLoad/tree/53114eac8e36f3d3a4f69ca6ee40a17e07299ef3`, 2020.

[Int22] Intel. *16 Intel® 64 and IA-32 Architectures Optimization Reference Manual*, chapter INTEL® TSX RECOMMENDATIONS, pages 503–532. Intel, 2022.

[KKL+07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. Proceedings of the linux symposium. In *Proceedings of the Linux Symposium Volume One*, pages 225–230, 2007.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[pie19] pietro borrello. Zombieload poc. `https://github.com/pietroborrello/RIDL-and-ZombieLoad/tree/4b2c858eb6948cf9fa3c43996ea5933f940e45f4`, 2019.

[PPL15] Nikolaos Pitropakis, Aggelos Pikrakis, and Costas Lambrinoudakis. Behaviour reflects personality: detecting co-residence attacks on xen-based cloud environments. *Int. J. Inf. Sec.*, 14(4):299–305, 2015.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 199–212. ACM, 2009.

[Shi21] Youngjoo Shin. Multibyte microarchitectural data sampling and its application to session key extraction attacks. *IEEE Access*, 9:80806–80820, 2021.

[SLM+19] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 753–768. ACM, 2019.

[UNR+05] Richard Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[vSMÖ+19] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 88–105. IEEE, 2019.

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 719–732. USENIX Association, 2014.