Reverse Branch Target Buffer Poisoning

José Luiz Negreira Castro de Oliveira
joselOliveira11@gmail.com
UFRJ

Diego Leonel Cadette Dutra
ddutra@cos.ufrj.br
UFRJ

***Abstract*—Memory corruption attacks are widely used to achieve remote code execution or privilege escalation in applications. To mitigate these attacks, the operational system usually randomizes the addresses of pages using Address Layout Randomization (ASLR). Using speculative execution, modern CPUs rely on branch predictors to choose the next instruction to be fetched in the CPU pipeline. This paper presents a new Spectre v2-based technique for abusing branch predictors to bypass ASLR on Intel CPUs. Our attack abuses the fact that not only can the attacker pollute the branch target buffer such as in a specter-like scenario, but victims can also trigger a branch misprediction in the attacker process, leading the attacker to speculatively jump to the same ASLR-protected address. Using a second cache side channel the attacker can then retrieve the address, completely bypassing the ASLR for the target process. Using a real 3rd generation Intel processor hosted on Google's cloud, we successfully recovered the victim's address 40% of the time, with 100% accuracy when recovered, with each execution taking around 15 seconds to complete.**

I. Introduction

Since the disclosure of the Spectre and Meltdown vulnerabilities [1], [2] many other vulnerabilities have been found inside of modern processors such [3], [4] , specially on the class of Microarchitecture Data Sampling (MDS). This class of attacks aims to abuse internal components of the CPU that operate in an insecure way when speculation [3] or page faults happens [4].

MDS are usually more reliable since they are able to read data directly from leaking components such as Line Fill Buffers (LFB) [3] or special registers inside CPU [5], therefore allowing the exploit to execute without the need of any information about the process being exploited or its addresses, that otherwise would be required when trying to exploit using meltdown or Spectre(v2) techniques. However the strong relation between virtual addresses and the success of this class of attacks may be abused in order to bypass security features such as ASLR, Position Independent Execution (PIE) or Kernel Address Layout Randomization (KASLR).

Spectre [1] describes how indirect calls affect the Branch Target Buffer (BTB) that is shared between processes using the same core. In the variant 2 of the attack, indirect call instructions are used to manipulate the BTB in order to achieve speculative arbitrary execution inside the victim context. In order to exploit successfully the attacker must know (at least partially) the source address of the vulnerable instruction and train the predictor to always jump to a specific destination inside the victim's context.

Many other attacks have abused the branch predictor in order to bypass randomization defenses [6], [7]. In JumpOverASLR the attack is focused on achieving a collision of the source address for the branch predictor and then measuring the time of execution of a specific function to check if a collision happened. With this exploit it was possible to leak the 12 randomized bits from KASLR, enough to bypass the mitigation, remaining only 9 bits needed to bruteforce for user mode randomization.

In Reverse Branch Target Buffer Poisoning (RBTBP) the indirect branch predictor is abused in order to leak the full 64 bit address of the call destination, a significant improvement when compared to JumpOverASLR [7]. The new attack also doesn't require any gadgets to be present on the victim's context, only the indirect call instruction to be called multiple times, all the gadgets required for the side channel are placed inside the attacker's context.

A. Tested Architecture

The Code was successfully tested on a Intel(R) Xeon(R) CPU @ 2.50GHz from Ivy Bridge available in google cloud. However literature points towards processors from Haswell family also being vulnerable, although we couldn't find one to test it yet. Previous generations have not been tested yet.

B. Threat model

Our Threat model assume that the attacker is able to execute unprivileged code on the target and tries to escalate to root user. To do so it exploits a victim's process running as root or with permission to sensitive files (e.g. another user process that can read files from /home). The Operating System (OS) has support for ASLR and the victim's program is compiled with PIE support and contains no address-leaking bugs. Therefore the base addresses of libraries and the binary sections are randomized in the virtual address space. We assume that the spy is able to execute on the same core as the victim that is a requirement for this exploit since both processes need to share the same target buffer, that exists only one per core. However there are many ways of forcing the coresidency within the same core as the victim [7]. As a requirement, the attacked application should contain a indirect call instruction, that can be executed multiple times. The spy processes controlled by the attacker then try to disclosure the victim's bases addresses and is able to use this information to exploit classic memory corruption bugs or even microarchitectural such as spectre v2. The attacker it's able to reverse engineering the binary and see it's instructions and partial addresses.

II. Background

A. Memory Corruption Exploits and mitigations

Many exploits rely on corrupting internal structures of the application, such as stack frames metadata or internal variables [8], [9] as result of improper input handling. These attacks are usually more common on applications written in low level languages, such as C.

In order to successfully exploit a memory corruption vulnerability e.g., a buffer overflow, the attacker must corrupt the return address pointer on the stack and make the function return to another section of the program. This was usually done by redirecting the program to execute the stack, where a sequence of specially crafted data to be interpreted as instructions (shellcode) was placed on the stack. When the return instruction is executed, the Instruction Pointer (IP) points to the shellcode placed on the stack to be executed next. This kind of attack can be used to achieve Remote Code Execution (RCE), or, if the target application runs at a higher privilege level or is the kernel itself, the exploit can lead to a privilege escalation scenario.

In order to prevent these kinds of attacks, many mitigations have been proposed and implemented, the most common being randomizations and data execution prevention. On modern Linux systems, the stack is marked as non executable, avoiding shellcodes placed on stack to be executed. As a bypass, attackers started using parts of the application's code itself as gadgets in attacks such as Return Oriented Programing (ROP) and Return to Libc (ret2libc) [9], [10] where the return instruction jumps to a location in the application or libraries that would do a malicious action such as opening a shell.

To mitigate Code Reuse attacks, operational systems started to support randomization of addresses in libraries with ASLR,in the binary itself with PIE and even the kernel with KASLR. Without knowing the base address of the code sections, it is harder for attackers to successfully create a code reuse attack chain.

ASLR and PIE randomize bits 12-41, meanwhile KASLR randomizes bits 21-29 [11], [7].

B. Virtual Memory

Physical memory management is an important task of modern OSs and processors, since they need to execute multiple programs simultaneously. One process cannot write in another process's memory because if that was possible a bugged program could accidentally crash another program by writing on the wrong address, or a malware would be able to read memory from other users on the same machine. Also, programs that were compiled to use the same addresses should be able to execute independently at the same time since the compiler is not able to know exactly what address are being used during the execution of the program. To achieve transparent memory management, the OS is responsible for translating virtual addresses chosen by the compiler into physical addresses used by the Random Access Memory (RAM) chip.

Pagination is used in order to isolate memory between processes. Each time a program accesses the memory the virtual address is translated to physical by querying the pagination table, a memory region owned by kernel responsible for storing pointers to the true memory. In order to reduce the amount of storage used, the pagination mechanism is divided into levels [12], therefore a single translation operation may require multiple memory accesses to the pagination table. To increase the performance of memory accesses, the results of the most recent translated addresses are cached in the Translation Lookaside Buffer (TLB), avoiding a new query each time a memory address is accessed. In addition whenever a switch occurs, such as changing execution from one process to another, the CR3 register that points to the base of the table is changed and the TLB must be flushed to invalidate its entries.

C. Side Channels

Instead of relying on a vulnerability on the algorithm, side channel attacks exploit the side effects of the execution in order to exfiltrate data. This side effects can be execution time [13], power consumption [14] or even wifi radio waves emitted by the memory bus [15].

In Microarchitecture side channels the attacker uses the state of internal components of the processor to infere what operations have been done inside an invisible context such as other process, kernel, Software Guard Extensions (SGX) or speculative execution. Some of the covert side channel can be the cache [16], [17],the TLB [18], [19] , row buffer in memory [20]. The flush+reload attack consists in measuring the time that a address takes to be read. To measure it accurately the RDTSC instruction is used to read the Time Stamp Counter register before and after the load operation. After the load, the address is invalidated in the cache with the CLFLUSH instruction. this avoids that a previous probing affects the next measure, so the probe only returns low values if that address was accessed recently by another context e.g. transient instruction.

Below is the function used for measuring the load time of a given address, adapted from flush+reload[16]. The LFENCE and MFENCE instructions works as serializing instructions, avoiding the undesired out of order execution in this section.

D. Speculative execution

Modern processors make a heavy use of pipeline architecture in order to increase the throughput of instructions. Some of the steps present in moderns pipeline are the instruction fetching decoding, reorder, schedule and execution [2]. The pipeline architecture allows the processor to execute multiple steps of the instruction in parallel, thus optimizing the usage of hardware by reducing the time spent waiting for the next instruction or data. Pipeline architecture works perfectly when the stream of instructions are sequential E.g. `add rax,rbx` . However, when the next instruction depends on the result of a still not executed instruction such as a conditional instruction `jz rip+0x10` the first step of the pipeline responsible from fetching the next instruction usually make a guess about the execution path taken therefore making a speculative execution on that address. If the

```
mfence
lfence
rdtsc
lfence
mov esi, eax
mov eax,[%1]
lfence
rdtsc
sub eax, esi
clflush [%1]
mfence
lfence
```

Listing 1: The TSC register can be used as fine grain memory performance measure

branch taken is proven correct later, the results are committed to the real registers, but if the path taken is wrong, the results are discarded yielding a result similar to stalling the fetch of the next instruction.

E. Branch predictors

Intel CPUs distinguish at least 3 types of predictors in [1]:

- Conditional branches occur after a pattern of `if (x) {do_something();}` being compiled. The task of the predictor is to decide whether the branch was taken or not. This mechanism is usually simple and depends on the frequency of the branch taken, so in order to train the predictor is only necessary to execute the desired path multiple times.
- Direct Calls and jumps e.g., `call rax` can be generated by calls of function pointers. Intel predictors uses the BTB to store the most recent direct and indirect branches [21].
- Indirect Calls and Jumps e.g., `call [rax]` are similar to the direct calls and uses the same BTB but the mechanisms of prediction are different.

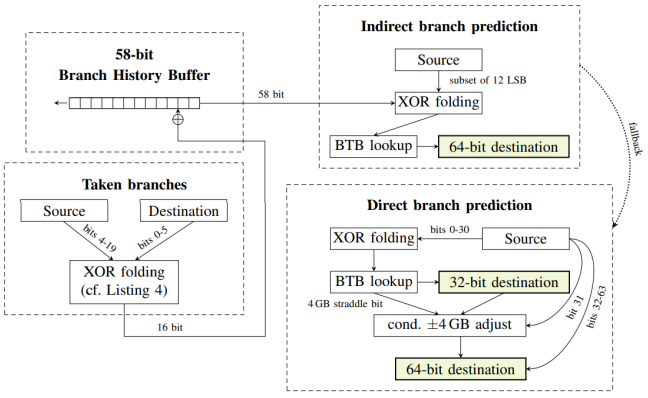

Figure 1: Mechanisms of branch prediction for direct, indirect and conditional branches. Extracted from Spectre [1].

The CPU also uses another buffer for predicting destinations of return addresses [22], [2].

It is worth noticing that the indirect predictor uses a subset of 12 Least Significant Bits (LSB) to resolve the full 64 bit address of the destination. Once the 12 LSB bits are not randomized by ASLR it is easy to force a collision on the source address using the indirect branch predictor, therefore this predictor is not suitable to attacks that exploit collisions on the predictor such as Jump Over ASLR [7].

F. Out Of Order Execution

Modern processors have more than one execution unit, allowing them to execute multiple instructions in a single clock cycle. In order to improve the usage of the execution units, the processor may change the order of some instructions when scheduling the operations, as long as the dependencies are respected. The processor must be aware of the three reorder hazards: Read After Write, Write After Read and Write After Write. These patterns contains dependencies that if reordered may change the output of the code.

G. Spectre

In Spectre variant 2 [1] shows how the branch predictor can be trained by an attacker to misspredict the destination address of a jump or call instruction. The attack consists of placing a branch instruction at a similar location so they are mapped for the same entry on the target buffer. An attacker can then jump multiple times to an address A in the attacker address space. When the attacked program reaches the branch, the predictor trained by the attacker will lead to execute speculatively at address A. Address A then should be chosen to be a spectre gadget, that would leak a secret into a side channel. An example of a spectre gadget could be `d = getenv_address[ secret[x] *4096 ]`, with the attacker having control, or being able to predict the variable x, so the libc would be used as side channel. In the original the following sequence of instructions were found at ntdll.dll with the dbx and edi controlled by the attacker:

```
adc edi,dword ptr [ebx+edx+13BE13BDh];
adc dl,byte ptr [edi];
```

Listing 2: Example of spectre gadget located at ntdll.dll

Since the attacker must know exactly the virtual address of the spectre gadget in order to exploit variant 2 of spectre is plausible to assume that randomization based techniques such as ASLR and PIE work as spectre v2 mitigations.

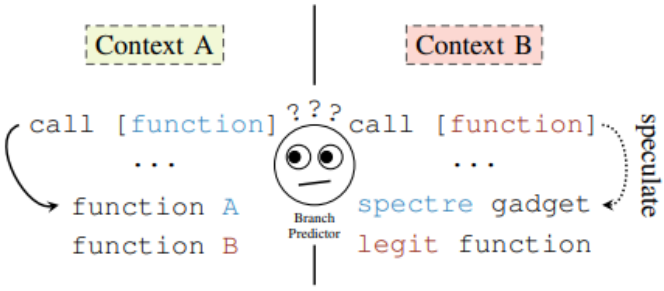

Figure 2: Context confusion on branches. Extracted from Spectre [1].

H. Branch predictor and ASLR

In Jump Over ASLR [7] the branch predictor is exploit in order recover addresses randomized by ASLR and KASLR. The attack consists on measuring the performance impact of BTB collisions. Since the direct branch predictor on Haswell architecture uses the bits 0-30 as source for the hashing [1], [7], [21] it's possible to create an executable that will try to align the 30 LSB of the jump instruction with another victim's jump instruction, then if they are truly 0-30 bits aligned, the attacker code will measure a longer time to execute the code block.

The kernel with KASLR mitigation enabled only randomizes bits 21-29 corresponding to the page directory, therefore, only 512 possibilities of addresses need to be tested to recover kernel address, making this attack very effective, bypassing KASLR in about 60 ms. In User-level ASLR however, Linux randomizes bits 12-40th [23] the attack can only recover bits 12-30, remaining 11 bits undisclosed. With 262.144 addresses (18 bits) needed to test and with test rate of 100 addresses per second according to the study, the attack on User-level applications would take roughly 40 minutes to leak the partial address.

III. Attack Overview

A vulnerable processor to RBTBP meets three requirements.

1) It must be vulnerable to branch target buffer poisoning (Spectre v2). In the meaning of calls and jumps executed in context A being able to interfere with executions on context B, given they share the same branch predictor components.
2) Must be able to execute Out of Order instructions, since the final exploit abuses the Out of Order Engine to execute incorrect branching paths for long periods of time.
3) Must use a poor and predictable hash function to allocate BTB entries. That means using few and known bits as input to the hash function that selects the index on BTB and not checking tags, allowing for an attacker to easily create a BTB collision, forcing both contexts to use the same entry. The predictor must also predicts to more bits than inputted.

Therefore, RBTBP brings a new branch predictor vulnerability to light. If Jump Over ASLR[7] exploits the difficulty of creating a BTB collision by bruteforcing all possible entry inputs to the hashing function, our attack uses the ease of creating collisions that will be predicted to a correct long address, leaking the full address with a (theoretical) complexity cost of 1.

According to the the Google Project Zero article[21] and our tests, Intel processors from the Haswell family meet these requirements. The indirect branch predictor uses only the first 12 LSB as input and predicts the full 64 bit address of the destination. Since the 12 LSB are not changed by ASLR, this predictor is perfect for the purpose of this attack.

This new attack follows the idea of spectre v2 of poisoning the BTB but instead of training the predictor to achieve transient execution on victim context, the attacker allows the victim to poison the BTB and then tries to discover what address was speculatively executed inside it's own context. Therefore there is no need for locating a spectre gadget inside victim's process, the only requirement for a vulnerable program is the presence of a indirect call instruction that can be executed multiple times.

A. Toy Example

Considering the following vulnerable section:

The steps needed for test the toy example are:

1) Placing an indirect call instruction e.g. `call [rdx]` on 12 LSB aligned address of another indirect call with the victim context. If the instructions have different sizes, the next instruction's address should match.
2) Placing the "leak gadget" on every possible destination address that victim could poison BTB to go to.
3) Letting the victim train the predictor
4) Passing "arguments" to leak gadget and then execute the call instruction.
5) Recovering results from speculative execution from the side channel.

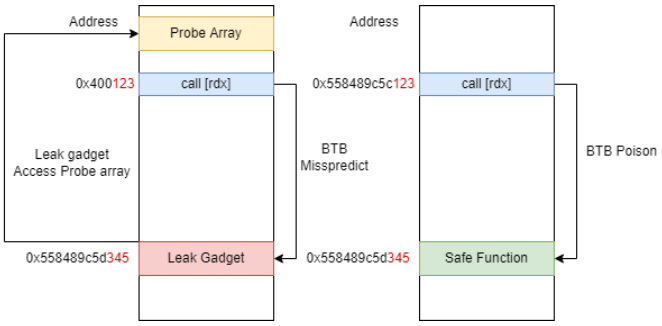

Figure 3: Memory Layout of the attack

The Leak Gadget is a piece of code that will execute speculatively and needs to inform to the attacker what is the address that it is executing.

This gadget Recives 2 arguments, the mask for the selected bit, present in cl register and the pointer to the probe array. First it loads the actual address of the first instruction on rax then selects the desired bit, if the selected bit is 0 it fetches the first element if probe array. If bit is 1 it fetches the 4096th element of probearray.

```
lea rax,[rip - 7] ;load current address
shr rax,cl        ;selects the bit using cl arg
and rax,1
shl rax,12        ;loads probearray
mov dl,[rsi+rax]  ;or probearray+4096
```

Listing 3: leaking gadget

After the speculative execution of the gadget it is possible to know if the selected bit of the address is 0 or 1 based on the time of response of the two elements in probearray[16].

B. Measuring the input and output size of the predictor

Just mount the toy example and change addresses to check if the gadget got executed, lol.

C. ASLR space range

In order to exploit this vulnerability all the possible destination locations should be filled with the leak gadget. However in a ubuntu18 system the address range of ASLR for text sections of the binary covers ,roughly from 0x550000000000 to 0x570000000000, that would require over 2TB of RAM memory, which is impracticable for current systems.

Although it's possible to test all the address spaces separately. First a block of size 4GB is created requesting mmap with anonymous option, then the page is filled it with gadgets and the attacker tries to speculate to see if the gadget is executed. If the gadget is not executed the 4GB window is slided down on the virtual address space with the mremap call, without changing the contents in the physical memory. The mremap call only changes the locations of the pointers to the physical memory inside the pagination tree, working as an extreme fast way of changing the virtual address of a block. This way in roughly 512 iterations to scan the complete address space, the gadget will collide with the destination predicted and will be executed, leaking the true destination address of the victim process.

In the first versions of the exploit, it was not possible to reliably execute the gadget even if the addresses were aligned because the translation for the address was not present in TLB. This was expected since the gadget should never execute in the normal program flow. It was required to fetch a single byte of the page containing the gadget, in order to cache the translation to the TLB and allowing it to execute[18], [19].

However, according with Intel Instruction Set Architecture (ISA) [24, Chapter 4.10.2.3], "The processor may cache translations required for prefetches and for accesses that are a result of speculative execution that would never actually occur in the executed code path". That means that even if the translation for the destination address is not present on the TLB the processor still may execute the pagewalk and cache the translation for that execution path. . In order to increase the transient execution time an Out of Order Execution technique was used. The first step in the instruction pipeline is the fetch and PreDecode [2]. In this stage the CPU frontend should fetch our gadget when the malicious call instruction is found. Only after computing the correct value for the call destination the path is fixed and the execution is redone. However is possible to increase the time that takes to correctly compute the destination address. When placing a sequence of chained memory reads before evaluating the destination it's possible to ensure that the CPU will take thousands of cycles to fix the path. Each of the pointers was flushed out of cached in the cycle before. The pointer chain looks like the following code:

```
mov rcx,%1        ;mask arg for gadget
lea rsi,[%2]      ;probe array ptr arg
lea rdx,[%0]
mov rdx,[rdx]     ;pointer chain
mov rdx,[rdx]
mov rdx,[rdx]
...
mov rdx,[rdx]
mov rdx,[rdx]
call [rdx]        ;mispredicts to gadget
lea rax,[rip - 7]
shr rax,cl
and rax,1
shl rax,12
mov dl,[rsi+rax]
```

Listing 4: Sequence of instructions fetched by the frontend

This pattern also has a second effect. Due to the dependency of read after write of the `mov rdx,[rdx]` instruction, it's impossible to perform any parallel execution on that sequence, but it allows the gadget to execute out of order, even before the call instruction is resolved.

```
mov rcx,%1        ;mask arg for gadget
lea rsi,[%2]      ;probe array ptr arg
lea rdx,[%0]
mov rdx,[rdx]     ;pointer chain
mov rdx,[rdx]
mov rdx,[rdx]
...               ;Out of order
lea rax,[rip - 7]
shr rax,cl
and rax,1
shl rax,12
mov dl,[rsi+rax]
...
mov rdx,[rdx]
mov rdx,[rdx]
mov rdx,[rdx]
call [rdx]       ;the execution path reverted
...
```

Listing 5: The gadget instructions already fetched can be executed out of order since there is no dependency

D. Results

Using 4 GB of memory is possible to test 1.048.576 addresses at once, but some iterations are needed to be reliable. One possible explanation is that caching the TLB may take multiple iterations, with each iteration caching part of the pagination table into the cache and finally the TLB. We tested with 512 iterations per block, with a result of at least 40% of success rate into breaking ASLR per execution. Each execution of the program takes about 15 seconds with a 100% of hit rate when the address is recovered.

IV. Discussion

A. Future works

The possibility of filling the entire address space with Copy on Write pages also was tested but with no speculative execution being measured. Changing the memory layout of the attack for this technique would allow to skip the part of remmaping of the attack and reduce the complexity to a true cost 1, however the implementation of this technique was not successful by the authors.

It was not possible to test the impacts on a hyperthread environment. The Processors used in the tests present on Cloud based provider may implement restrictions on the scheduling on the same core as mitigations for new MDS class attacks.

It looks like it's possible to extend the attack to architectures that uses more than 12 bit as input for the hashing function of BTB. However before this attack can be implemented it's necessary to locate a collision in BTB using collision attacks as described before [7].

B. Mitigations

1) Avoiding Spectre v2: Several mitigations for avoiding spectre were proposed[25], such as preventing branch poisoning with Indirect Branch Restricted Speculation (IBRS),Single Thread Indirect Branch Prediction (STIBP), Indirect Branch Predictor Barrier (IBPB) that prevents branch poisoning across multiple contexts or retpotlines, that changes jumps and calls instructions to returns. Instructions such as lfence and mfence before jumps are not effective to mitigate RBTBP since there is no speculation based execution on the victim side.

2) Enhancing hashing functions on branch predictors: The simple input of 12 LSB to the hashing function of the BTB makes very easy to force collisions with other contexts since none of those inputs are randomized. When a larger input is used on the hashing function it forces the attack to at least bruteforce each possible input to detect a collision with the victim context, greatly increasing the amount of time to perform the full attack.

3) Reducing the amount of destination bits stored: Other mitigation is to use the partial address of the source as destination, with the predictor storing only the least significant half of the destination address. This ensures that only half of the bits can be leaked if the destination is disclosed. The indirect branch predictor in Haswell architectures uses bits 0-30 as input for the BTB and reuses source address 31-63 to create the full destination address. Our tests also showed that more recent Intel CPU deprecated the usage of the vulnerable indirect branch predictor and started to use the "direct" predictor for both branches, although still vulnerable to source collision based attacks [7] it limits the amount of bits predicted to only 30, rendering RBTBP useless.

Acknowledgment

The authors would like to thanks Epic Leet Team (ELT) Capture The Flag Team and Security Incident Response Group (GRIS) from Rio de Janeiro Federal University (UFRJ) for the support during the research and revision phase of this project.

References

[1] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[3] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *S&P*, May 2019.

[4] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *CCS*, 2019.

[5] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "CrossTalk: Speculative Data Leaks Across Cores Are Real," in *S&P*, May 2021, intel Bounty Reward. [Online]. Available: Paper=https://download.vusec.net/papers/crosstalk_sp21.pdfWeb=https://www.vusec.net/projects/crosstalkCode=https://github.com/vusec/ridlPress=https://bit.ly/3frdRuV

[6] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, *Speculative Probing: Hacking Blind in the Spectre Era*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1871–1885. [Online]. Available: https://doi.org/10.1145/3372297.3417289

[7] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[8] A. One, "Smashing the stack for fun and profit," *Phrack Magazin*, vol. 7, no. 49, 1996.

[9] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, mar 2012. [Online]. Available: https://doi.org/10.1145/2133375.2133377

[10] c0ntex, "Bypassing non-executable-stack during exploitation using return-to-libc."

[11] J. Ganz and S. Peisert, "Aslr: How robust is the randomness?" in *2017 IEEE Cybersecurity Development (SecDev)*, 2017, pp. 34–41.

[12] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1st ed. Arpaci-Dusseau Books, August 2018.

[13] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *12th USENIX Security Symposium (USENIX Security 03)*. Washington, D.C.: USENIX Association, Aug. 2003. [Online]. Available: https://www.usenix.org/conference/12th-usenix-security-symposium/remote-timing-attacks-are-practical

[14] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: Software-based Power Side-Channel Attacks on x86," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

[15] M. Guri, "Air-fi: Generating covert wi-fi signals from air-gapped computers," 2020.

[16] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[17] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," 2016.

[18] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security*, Aug. 2018, pwnie Award Nomination for Most Innovative Research. [Online]. Available: Paper=https://download.vusec.net/papers/tlbleed_sec18.pdfSlides=https://www.usenix.org/sites/default/files/conference/protected-files/security18_slides_gras.pdfWeb=https://www.vusec.net/projects/tlbleedCode=https://github.com/vusec/tlbkitPress=https://goo.gl/eepq1y

[19] ——, "TLBleed: When Protecting Your CPU Caches is not Enough," in *Black Hat USA*, Aug. 2018. [Online]. Available: Slides=https://i.blackhat.com/us-18/Thu-August-9/us-18-Gras-TLBleed-When-Protecting-Your-CPU-Caches-is-Not-Enough.pdfWeb=https://vusec.net/projects/tlbleed

[20] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "Drama: Exploiting dram addressing for cross-cpu attacks," 08 2016.

[21] "Reading privileged memory with a side-channel," https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, accessado em: 03/01/2022.

[22] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *Proceedings of the 12th USENIX Conference on Offensive Technologies*, ser. WOOT'18. USA: USENIX Association, 2018, p. 3.

[23] H. Marco Gisbert and I. Ripoli, "On the effectiveness of full-aslr on 64-bit linux," Nov. 2014, in-depth Security Conference 2014 (DeepSec) ; Conference date: 18-11-2014 Through 21-11-2014. [Online]. Available: https://deepsec.net/archive/2014.deepsec.net/index.html

[24] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3*. Santa Clara, USA: Intel Corporation, 2016, iSBN 325384-060US.

[25] *Speculative Execution Side Channel Mitigations*. Santa Clara, USA: Intel Corporation, 2018, iSBN 336996-003.