GhostBusters: Mitigating Spectre Attacks on a DBT-Based Processor

Simon Rokicki
Univ Rennes, INRIA, CNRS, IRISA

***Abstract*—Unveiled early 2018, the Spectre vulnerability affects most of the modern high-performance processors. Spectre variants exploit the speculative execution mechanisms and a cache side-channel attack to leak secret data. As of today, the main countermeasures consist of turning off the speculation, which drastically reduces the processor performance. In this work, we focus on a different kind of micro-architecture: the DBT based processors, such as Transmeta Crusoe [1], NVidia Denver [2], or Hybrid-DBT [3]. Instead of using complex out-of-order (OoO) mechanisms, those cores combines a software Dynamic Binary Translation mechanism (DBT) and a parallel in-order architecture, typically a VLIW core. The DBT is in charge of translating and optimizing the binaries before their execution. Studies show that DBT based processors can reach the performance level of OoO cores for regular enough applications. In this paper, we demonstrate that, even if those processors do not use OoO execution, they are still vulnerable to Spectre variants, because of the DBT optimizations. However, we also demonstrate that those systems can easily be patched, as the DBT is done in software and has fine-grained control over the optimization process.**

I. Introduction

The growing importance of data and the increasing reliance on computer systems have made cybersecurity a primary concern and a very active domain of research. Usually, security vulnerabilities occur at a software level (e.g., the classical buffer overflow). Recently, a new kind of vulnerability caused by the hardware has been unveiled. The Spectre vulnerability exploits speculative execution and cache side-channel attacks to access private data on a remote system [4]. After the first unveiling in the early 2018s, several variants of Spectre have been added in the set of speculation based attacks [5]–[7].

As out-of-order and speculative execution is a crucial feature for high-performance computing, almost every high-end processor is sensible to this vulnerability, and dealing with these vulnerabilities is a challenge for micro-architecture researchers. Indeed, current countermeasures prevent any speculation on sensible Spectre patterns, at the expense of processor performance.

The above mentioned out-of-order (OoO) and speculative execution introduce significant overhead in the energy consumption of the system. For embedded systems, where energy efficiency is a concern, other kinds of micro-architectures are used. For example, micro-architectures based on Dynamic Binary Translation (DBT) offer high-performance computing on data-intensive applications, while consuming less energy than OoO processors. The idea of DBT based processors is to dynamically translate the (sequential) binaries of the executed application into explicitly parallel binaries, which are executed by an in-order processor. There are several examples of such micro-architectures: Transmeta Crusoe and Efficeon processors [1], NVidia Denver [2] and the academic tool Hybrid-DBT [3].

In this paper, we demonstrate that, even if such micro-architectures are based on in-order architectures, they are sensible to Spectre vulnerability. Indeed, the speculation is done by the DBT engine, which re-orders instructions to increase performance. We have built several proof-of-concept attacks for two variants of Spectre and validated them on NVidia Denver processor and Hybrid-DBT. We have also built a countermeasure for Spectre, which has been implemented on Hybrid-DBT.

More precisely, the contributions of this work are i) a proof-of-concept of Spectre attacks on DBT based processors and ii) the presentation of a simple countermeasure, which is made because such micro-architectures offer precise control over the speculation process. This countermeasure is an update on the DBT software and causes no real slowdown in the benchmark applications studied.

The rest of this paper is organized as follow: in Section II, we provide all the necessary background on speculation techniques, Spectre, and on the principle of DBT based processors. Section III presents the two variants of Spectre which have been tested on DBT based processors, and Section IV presents the countermeasures applied in Hybrid-DBT. Finally, Section V presents the experimental study conducted, and Section VI present the related work.

II. Background

In this section, we provide all the necessary background on speculative Out-of-Order execution and the Spectre vulnerability. We also quickly introduce DBT based processors.

A. *Speculative execution and Spectre vulnerability*

Most modern high-performance processors use speculative execution to increase the number of instructions executed in parallel. The principle is to run some instruction on unused execution units without knowing for sure if it has to be executed (e.g., branch prediction), if its operands are valid (e.g., value prediction) or if another instruction should have been executed before (e.g., memory dependency speculation). Speculative execution is based on complex hardware mechanisms, which ensure that a misspeculated instruction never affects the architectural state of the processor. Because

978-3-9819263-4-7/DATE20/©2020 EDAA

```c
char buffer[size];
char arrayVal[256*128];

if (index < size){
    char a = buffer[index];
    char b = arrayVal[a * 128];
}

//Extracting the value from the cache
inspectCache();
```

Fig. 1. Example code of Spectre v1

of the page limitation, we do not provide more details on the complex mechanics involved in speculative out-of-order micro-architecture.

The Spectre vulnerability unveiled early 2018 takes advantage of these speculation mechanisms. The basic idea is that, even if a misspeculated instruction does not modify the state of the architecture, it may affect the micro-architecture. This change in the micro-architecture can then be moved toward the architecture using side-channel attacks. In the example of Spectre, the leakage comes from the cache: the attacker modifies the cache state function of the secret value (i.e., access a memory location whose address depends on the secret value) and uses performance counters to detect which memory location is in the cache.

Let us consider the code on Figure 1. Extracting a secret value using the Spectre vulnerability requires three steps:

1) The attacker trains the branch predictor to enter inside the if condition: he/she executes the code using indexes that are within the array boundaries.
2) The attacker flushes the data cache and generates an index which points to the secret value (i.e., the index value is the difference between the address of the secret value and the address of the buffer). During the execution, the branch predictor predicts that the execution enters the if condition and, consequently, the processor starts executing these instructions. It reads the secret value, computes an index depending on this secret and uses it to access a given memory location in `arrayVal`.
3) The processor commits the branch instruction, realizes that these instructions should not be executed and drop their results. However, the cache has been affected by the speculative execution.
4) The attacker inspects the cache by measuring the time required to read a value of `arrayVal`. For one of these values, the access time is much shorter, which means that the value was already in the cache: the attacker knows the secret value.

As the memory access performed to read the secret is done speculatively, any exception raised when the value is accessed are dropped when the instruction is dropped (e.g., exceptions indicating that the process has no read permission at this memory location). The example in Figure 1 exploits the branch prediction mechanism, but there exist variants based on various speculation mechanisms.

B. DBT based processors

Superscalar OoO processors use sophisticated hardware to dynamically re-schedule instructions and to speculate on branches or dependencies. If these mechanisms enable high-performance on a broad range of applications, it also consumes much energy. For embedded systems where energy efficiency is a primary concern, it is possible to use DBT based processors, which offer performance comparable with OoO processors but at a lower energy expense.

The idea behind DBT based processors is to use a simple in-order superscalar processor combined with a Dynamic Binary Translation engine. During the execution, the DBT layer generates binaries optimized for the in-order architecture: it builds execution traces, super-blocks or hyper-blocks and uses these structures to re-schedule the instructions. The objective is to improve the binaries with knowledge of the in-order processor used.

There are several examples of such processors: Transmeta Crusoe and Efficeon processors execute x86 on a VLIW architecture [1], NVidia Denver and Carmel processors execute Arm-v8 binaries in a superscalar in-order core [2]. Finally, Hybrid-DBT is an open-source, DBT based processor which executes RISC-V binaries on a VLIW architecture [3]. For all those processors, a software thread analyzes, translates and optimizes the binaries to increase the performance on the in-order architecture.

Some studies have shown that, with perfect static schedules, it is possible to reach 90% of the performance level of an Out-of-Order processor using the same number of computing units [8]. Moreover, as the DBT based processor do not use complex hardware mechanisms to re-order instructions and speculate, they can offer more execution units. As an example, the last Carmel core from NVidia is a 10-wide superscalar processor.

In the next section, we present how DBT based processors are subject to the Spectre vulnerability because of speculation made by the DBT engine. However, we also demonstrate that, as the DBT is done in software, it can be patched to be resilient to the vulnerability.

III. Spectre on DBT based processor

The first contribution presented in this paper is to demonstrate that DBT based processors are vulnerable to some variants of the Spectre vulnerability, even if the underlying hardware executes instructions in their sequential order. Indeed, when the binaries are translated and optimized, the DBT engine introduces speculative instructions.

In this section, we present two variants of Spectre that works on both NVidia Denver processor and Hybrid-DBT. As the speculation process is slightly different in DBT based processors, the attacker code has been modified to exploit these mechanisms.

A. Speculation during trace-based scheduling

Among the optimizations applied by the DBT engine, the most important one is *block construction*. The execution is profiled, and the outcome of frequently executed branches is collected. Then, the DBT engine merges basic blocks and

duplicates instructions to create super-block, hyper-block, or execution traces. Inside those blocks, the DBT engine performs instruction scheduling and register allocation. If an execution unit is available, it may move some instructions before a conditional branch, allocating a hidden register for storing the result (i.e., a register not defined in the ISA). During the execution, if the conditional branch has been predicted correctly, the execution is slightly faster, as some instructions are already executed. If the conditional branch has been miss-predicted, the result of the instruction is ignored, and the execution continues. As for out-of-order processors, it is possible to use cache side-channel attacks to leak the result of the speculative instruction.

On the example given in Figure 1, the attacker can execute the binary many times using an index strictly lower than the array size. There, the DBT engine observes that the conditional branch is almost always not taken. Consequently, it decides to merge the block located before the `if` and the `then` block. This newly created block contains both the comparison, the conditional branch and the two memory accesses used for the Spectre attack. During instruction scheduling, those two memory accesses may be scheduled before the conditional branch, enabling the Spectre attack. This variant of Spectre corresponds to Spectre v1.

B. Speculation with the Memory Conflict Buffer

Another challenging point in DBT based processor is to handle memory disambiguation. At the binary level, the DBT engine has no access to memory addresses (only register plus offset) and it is challenging and time-consuming to prove that two memory operations are independent. However, DBT engines can use memory dependency speculation to circumvent this problem: at schedule time, it speculates that there are no dependencies between two memory operations, and schedules accordingly. Those speculative memory operations are clearly identified in the binaries (i.e., using a distinct opcode in the VLIW ISA). At execution time, dedicated hardware is used to store and compare the addresses of speculative memory operations. If the two addresses are different, the speculation was correct, and the execution continues. If the two addresses are the same, the memory accesses were dependent, and the execution is currently in an illegal state. In this situation, there are two possibilities: either the DBT engine has already generated recovery code for this situation, and it can be executed, or the execution is interrupted until such code is generated.

This speculation mechanism, which is close to Load/Store Queues in OoO processors, has been first introduced by Gallagher et al. and mainly focused on static compilation [9], [10]. This speculation technique is used in Transmeta Crusoe and Efficeon processors [1], in NVidia Denver [2] and in Hybrid-DBT [11].

Here again, it is possible to speculatively read a value in memory and to extract it using a cache side-channel attack. Let us consider the example code in Figure 2, and more specifically, the situation where $i = j = 0$. If all instructions are executed in their sequential order, the `addrBuf` ultimately contains a valid index in its first location, which is then stored in variable `a`. This value `a` is then used to access the array `buffer`.

```
int  addrBuf[8];
char buffer[size];
char arrayVal[256*128];

addrBuf[i] = &secret - &buffer;
addrBuf[j] = ... ; //long computation

int a = addrBuf[0];
char b = buffer[a];
char c = buffer[b*128];

//Extracting the value from the cache
inspectCache();
```

Fig. 2. Example code of a variant of Spectre exploiting memory dependency speculation.

On a DBT based processor using memory dependency speculation, the order of memory operations may be changed. Indeed, as the second write on `addrBuf` requires long computations, all the subsequent load operations may be scheduled before it. Consequently, at run-time, the processor loads an incorrect index in `a`, uses it to read the secret value, which is then stored in `b`. This value is then used to compute an address, which is necessary to leak the value using cache side-channel attacks.

After those operations, the store instruction on array `addrBuf` is executed, and the dedicated hardware which compares addresses realizes that the memory location has been accessed before it was written (i.e., realizes that the value of `a` was incorrect). Consequently, it rolls back and re-execute the instruction correctly, but the value has already been leaked using the cache.

A similar attack can be performed using the Load/Store Queue of an OoO processor. Spectre variant 4 exploits this kind of mechanisms.

IV. Mitigation of Spectre Vulnerability

As presented in the previous section, DBT based processors are also vulnerable to Spectre variants. However, they provide additional control in the speculative execution. Indeed, the optimization process is performed in software and can be patched to insert countermeasures in the instruction scheduling.

In the following, we present the modification done in the DBT engine to i) dynamically detect a possible Spectre vulnerability in the optimized binaries and ii) generate Spectre-safe binaries without sacrificing performance. Those modifications have been implemented in the open-source Hybrid-DBT simulator.

A. Detecting Spectre patterns

The first objective is to detect the Spectre pattern in binaries. After Spectre unveiling, several attempts have been made to detect this pattern in compilers and to insert fence instructions

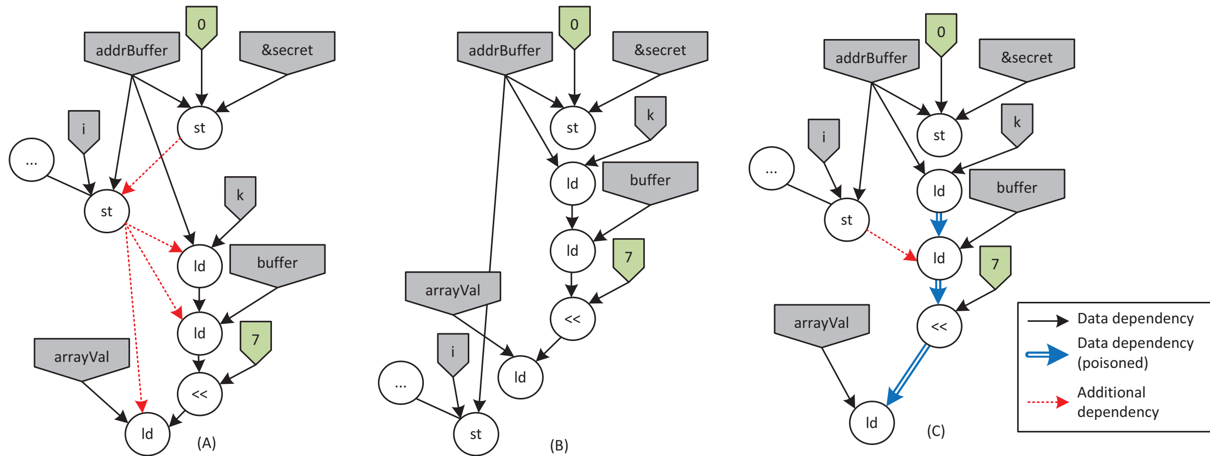

Fig. 3. Data-flow graph of a Spectre v4 attack code. Part (A) of the figure represents the original data-flow graph with the dependencies between memory operations. Part (B) of the figure corresponds to the most aggressive version, where the DBT engine speculate on all memory operations and schedules consequently. Part (C) represents the graph used in our approach: output of speculative loads have been poisoned and the memory operations using a poisoned address are constrained.

to secure the binaries [12]. However, as OoO processors automatically speculate over all branches, the pattern can be hidden behind complex control-flow operations. Consequently, such a tool has to analyze the whole application.

In our approach, the problem is much simpler: before performing instruction scheduling, the DBT engine has access to an Intermediate Representation containing all the instructions to schedule. No speculation can be done outside the scope of a single IR block. Temporary values are dropped at the end of the IR block and cannot be accessed from somewhere else. Consequently, we only have to search for the Spectre pattern inside a single IR block.

For detecting the Spectre pattern, we use a poisoning mechanism: all the values manipulated on the data-flow graph may be *poisoned* by a speculative instruction. The goal of the analysis is to detect the load pattern of the cache side-channel attack, which is a speculative load instruction using a poisoned value as an address.

The analysis consists of going through all instructions of an IR block applying the following rules:

- A speculative instruction generates a poisoned value. Speculative instructions can be either load instruction moved before[1] a conditional branch or load instruction moved before a memory write;
- If an instruction uses a poisoned value as an operand, it generates a poisoned value;
- If a speculative memory instruction uses a poisoned value as an address, then it may leak a value through a side-channel attack. Consequently, we mark the instruction to ensure that it cannot be scheduled speculatively.

Figure 3 illustrates this analysis. Part (A) of the figure is the data-flow graph with no speculation: we can note the dependencies between the store instruction and the different load instructions. Part (B) of the figure is the same data-flow graph with the memory speculation enabled: the DBT engine has removed those dependencies to generate a denser schedule. This version may lead to a Spectre vulnerability. Part (C) shows the result of the poisoning analysis: the blue, doubled line arrows correspond to poisoned values.

B. Constraining the schedule

When the Spectre pattern has been identified, we constrain the schedule so that no data could leak through it. The idea here is to stop the speculation. On Intel processors, this can be achieved using Fence instructions, which stalls the instruction fetch until all speculative instruction are committed. When using a DBT based processor, we have fine-grained control over the instruction scheduling. In Hybrid-DBT, a data-flow graph encoded in the IR can use some control dependencies to constrain the schedule. If there is such dependency between instruction A and B, the instruction scheduler ensures that instruction B is scheduled after instruction A. When a Spectre pattern is detected using poisoning, the DBT engine inserts a control dependency between this instruction and the one which causes the speculation, preventing the leakage.

In part (C) of Figure 3, we can see that a control dependency (with a red dashed line) has been added between the store instruction and the first speculative load which uses a speculative value as an address.

The use of this fine-grained control reduces the cost of the countermeasure, as it only constrains the risky instructions and those depending on it. If there are other instructions in the IR block, they can be scheduled speculatively.

V. Results

In this section, we present the experimental study performed. There are two objectives in this study: first, we demonstrate that the Spectre variants described in section III are effectively working on the architecture tested and that the countermeasure described in section IV effectively mitigates them; then we measure the slowdown caused by our countermeasures on a set of benchmark applications.

[1]As we work in a data-flow graph, the concept of place of an instruction has no meaning. When we say *moved before*, we means that the dependency between the conditional branch and this load instruction has been removed to possibly generate a more efficient schedule.

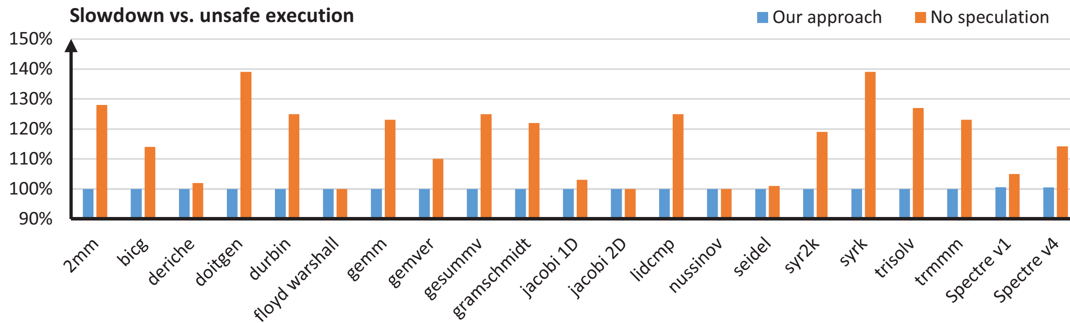

Fig. 4. Slowdown caused by the different countermeasure tested, compared with unsafe execution. First value corresponds to the slowdown in our situation. Second value is the slowdown when speculation is turned off. Lower values are better.

A. Proof of concept and countermeasure

The two variants of Spectre described in section III were implemented both in Armv8 and in RISC-V (using the rv64im ISA). The Arm version is tested on the NVidia Jetson TX2 developer board, which features two Denver cores and two standard OoO cores. The RISC-V version is tested on Hybrid-DBT simulator [3]. For the two versions of each variant, we demonstrate that we can read the value of a memory location which should not be readable.

For flushing the cache, the Arm version uses a dedicated instruction whereas the RISC-V version has to perform an explicit line by line flush, which slows down the attack. To perform the cache side-channel attack, we need to measure the memory accesses time and determine whether it is a hit or a miss. In the Arm version, it is done using the Performance Monitor Unit defined in Arm-v8, which is accessible from the user-space of Linux. In the RISC-V version, we read the state register, which counts the number of cycles. It is also interesting to note that performing the cache side-channel attack is more straightforward on DBT based processor than on OoO cores. Indeed, DBT based use in-order execution, where the timing is more stable than for OoO cores, which simplifies the distinction between hits and misses.

We have also implemented the proposed countermeasure in Hybrid-DBT and ensured that the same binaries are no longer subject to the Spectre vulnerability.

This study demonstrated that both NVidia Denver and Hybrid-DBT are vulnerable to the two Spectre variants described in III. We also demonstrated that, with a simple software update on Hybrid-DBT software, the system is no longer vulnerable to these two variants.

B. Measuring the loss of performance

The second experiment consists in measuring the slowdown caused by the countermeasure. The objective of this experiment is to demonstrate that, while making the core resilient to Spectre variants, our countermeasure does not affect the performance of applications which are usually executed on a DBT processor. As explained in the related work [3], DBT processors are more efficient on data-intensive applications. Consequently, we based our experiments on this kind of application: we have executed a set of application from Polybench using different parameters for the countermeasures. For each execution, we collect the execution time and measure the performance degradation caused by the countermeasure. Besides, we have executed the two proof-of-concept attacks presented earlier: application `Spectre v1` is the variant based on the trace construction, application `Spectre v4` is the one based on memory dependency speculation.

Results of this experiment are represented in Figure 4. The `baseline` is the unsafe execution (i.e., without any countermeasure). The data labeled `our approach` is an implementation of the countermeasure described in section IV. The one labeled `No Speculation` is a safe execution where the two kinds of speculation are turned off in the optimization design (i.e., a naive way to protect against Spectre variants).

We can see that on most of the application studied the countermeasure does not cause any slowdown. On the contrary, the simple countermeasure, where the speculation is turned off in the DBT engine, has a significant impact on performance, increasing the execution time by 16% on average. There are two possible explanations for the absence of performance degradation with our approach: either it is caused by the detection of the Spectre patterns which eliminates most of the speculation done, either it is caused by the fine-grained mitigation performed on the binaries. To answer this question, we did a third experiment where we added a `fence` whenever the Spectre pattern is detected. Here again, the countermeasure does not impact the execution time, which means that the Spectre pattern is not commonly seen on the binaries. Finally, we have manually modified the matrix multiplication benchmark to insert the Spectre pattern and measure the impact of the countermeasure. We have modified the way 2D arrays are represented, selecting the one based on arrays of pointers. Consequently, there are much more double indirection accesses, which increase the occurrence rate of Spectre patterns. On this modified application, our fine-grained countermeasure increases the execution time by 4% while the one based on a `fence` increases the execution time by 15%. This highlight that, when the Spectre pattern is identified, fine-grained mitigation has a smaller impact on performance.

VI. Related work

Security has long been a software issue. However, since the early 2018s and the unveiling of Spectre and Meltdown attacks, the community realized that hardware could add

security vulnerabilities [4]. After the initial Spectre unveiling, which is centered around branch prediction and branch target predictions, other variants have been discovered and added to the set of speculation based attacks. Some exploit the load/store queue, and others exploit the state registers of the floating-point unit. More recently, the Spoiler attack exploits the Memory Order Buffer to leak information about physical address mapping [5]. In this work, we have only considered the Spectre variants based on branch prediction and load/store queue because they have their equivalent in a DBT based processor. The current version of Hybrid-DBT has no comparable mechanisms to branch target prediction, but such a feature could be added in the trace construction mechanism of an industrial-strength DBT processor.

Right after the Spectre unveiling, several countermeasures have been proposed [6], [13]. The most straightforward consists of inserting `fence` instructions to prevent speculation. However, this instruction only acts as a memory barrier, and its exact behavior is not clearly defined. There were also propositions for removing or reducing the precision of timers, breaking the cache side-channel attacks, but some studies have shown that precise timers can be built using another thread. Finally, several JIT tools choose to generate only branch-less masking, which protects against out-of-bound accesses using only arithmetical operations.

Mcilroy et al. [13] introduced a more precise definition of Spectre vulnerabilities, with the definition of the architectural state and the micro-architectural state. The first corresponds to registers defined in the ISA, and the latter represents implementation-dependant registers. Typically, the register file is included in the architectural state while the state of the data cache is in the micro-architectural state. Mcilroy et al. also listed operations for reading and modifying the micro-architectural state, like cache side-channel attacks.

Fadiheh et al. [7] proposed a model to verify whether an architecture is vulnerable to Spectre or not. Their models, which depends on the hardware implementation, detect whether a change in the micro-architectural state can be moved to the architectural state. The analysis is expensive and has only been applied on a simple pipelined processor, highlighting a new variant of Spectre, which exploits pipelined data caches. Spectector from Guarnieri et al. uses symbolic execution to detect whether an application is vulnerable to Spectre [14].

Finally, OO7 [12] uses a tainting analysis to detect whether a dangerous value (i.e., a value which can be controlled by an attacker) can be used as an address for a speculative load. If such a pattern is detected, a `FENCE` instruction can be inserted to make the binaries safe. Their tainting analysis is similar to the proposed approach. However, the particular case of DBT processors simplifies the problem as we know exactly the scope of the speculation. Indeed, the DBT engine only speculates inside the block boundaries. Consequently, we only have to analyze these instructions, while OO7 has to analyze the whole binaries as Spectre can be exploited even through function calls.

VII. Conclusion

Even if DBT based processors are vulnerable to Spectre variants because of the dynamic optimization layer, we have demonstrated that they can be patched to remove the vulnerability. Our experimental study shows that our countermeasure does not affect the performance of the system. Future work will be focused on investigating other sources of speculation in the DBT engine to ensure that any variant of Spectre can be mitigated using this kind of mechanisms. Beyond the speculative execution, we also have to make sure that the optimization decision made in the DBT engine does not leak information on secret data.

Acknowledgements

The author would like to thank Thibaud Balem, Thais Baudon, Marco Freire, and Dylan Marinho for their contribution to the project.

References

[1] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing™ Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. IEEE Computer Society, 2003, pp. 15–24.

[2] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman, "Denver: Nvidia's First 64-bit ARM Processor," in *IEEE Micro*, vol. 35, Mar. 2015.

[3] S. Rokicki, E. Rohou, and S. Derrien, "Hybrid-DBT: Hardware/Software Dynamic Binary Translation Targeting VLIW," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[4] 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:1801.01203 [cs]*, Jan. 2018.

[5] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar, "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks," *arXiv:1903.00446 [cs]*, Mar. 2019.

[6] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," *arXiv:1807.03757 [cs]*, Jul. 2018.

[7] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz, "Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2019, pp. 994–999.

[8] D. S. McFarlin, C. Tucker, and C. Zilles, "Discerning the Dominant Out-of-order Performance Advantage: Is It Speculation or Dynamism?" in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 241–252.

[9] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W.-m. W. Hwu, "Dynamic Memory Disambiguation Using the Memory Conflict Buffer," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VI. New York, NY, USA: ACM, 1994.

[10] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, Ben-Chung Cheng, P. R. Eaton, Q. B. Olaniran, and W.-W. Hwu, "Integrated predicated and speculative execution in the IMPACT EPIC architecture," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, Jul. 1998, pp. 227–237.

[11] S. Rokicki, E. Rohou, and S. Derrien, "Aggressive Memory Speculation in HW/SW Co-Designed Machines," in *DATE 2019 - 22nd IEEE/ACM Design, Automation and Test in Europe*. Florence, Italy: IEEE, 2019.

[12] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "Oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis," *arXiv:1807.05843 [cs]*, Jul. 2018.

[13] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv:1902.05178 [cs]*, Feb. 2019.

[14] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "SPECTECTOR: Principled Detection of Speculative Information Flows," *arXiv:1812.08639 [cs]*, Dec. 2018.