Three Sidekicks to Support Spectre Countermeasures

Markus Krausz[1], Jan Philipp Thoma[1], Florian Stolz[1], Marc Fyrbiak[2], Tim Güneysu[1]
[1]*Ruhr-University Bochum, Germany*, firstname.lastname@rub.de
[2]*Emproof, Bochum, Germany*

***Abstract*—The Spectre attack revealed a critical security threat posed by speculative execution and since then numerous related attacks have been discovered and exploited to leak secrets across process boundaries. As the primary cause of the attack is deeply rooted in the microarchitectural processor design, mitigating speculative execution attacks with minimal impact on performance is far from straightforward. For example, various countermeasures have been proposed to limit speculative execution for certain instruction patterns, however, resulting in severe performance overheads. In this paper, we propose a set of code transformations to reduce the number of speculatively executed instructions and therefore significantly reduce the performance overhead of various countermeasures. We evaluate our code transformations combined with a hardware-based countermeasure in gem5. Our results demonstrate that our code transformations speed up the secure system by up to 16.6%.**

***Index Terms*—Spectre, Transient Execution, Branch Prediction**

I. Introduction

The ability to predict the control flow is one of the most important performance enhancement techniques deployed in modern CPUs. Due to pipelining, superscalarity, and out-of-order execution, CPUs can execute many instructions between the point in time when a branch is first fetched and the time when it is finally executed. If a branch condition depends on data that needs to be loaded from memory, hundreds of cycles can pass until the data is available and the branch can be resolved. During that time, modern CPUs make an educated guess where the branch might go based on previous executions and continue execution at that location. Instructions that are executed during speculation are not visible from the architectural CPU state. Only when the branch prediction was correct, the CPU can commit the speculative instructions and continue execution without any performance loss. On a misprediction, however, the CPU rolls back the incorrectly executed instructions and continues execution from the wrongly-predicted branch.

Though speculatively executed instructions do not leave any architectural traces, they do leave traces in the microarchitecture, i.e., the cache, the TLB and CPU internal buffers.These microarchitectural leakages have been exploited in several so-called transient-execution attacks [2]. One of the first and most prominent transient execution attacks is Spectre-v1 [6], where the attacker maliciously trains the branch predictor to make a wrong branch prediction. The attacker can then lead the CPU to make a speculative memory access to a potentially arbitrary memory address. Though the result of this access is never visible to the architectural level, the attacker can exploit microarchitectural side-channels to leak the data. Many different countermeasures have been proposed, which vary in strategy and performance overhead.

In this paper, we propose a complementary direction of research by exploring options to limit the amount of speculation that needs to occur within the CPU. By that we inherently limit the attack surface for speculative execution attacks and reduce the performance overhead of speculation-limiting countermeasures. To achieve this, we investigate modifications in software, the compiler, and on the architectural level. In particular, we introduce compiler transformations that allow branches to be resolved sooner, thus narrowing the speculative windows and resulting in improved performance and smaller transient attack windows. Moreover, we explore the effect of linearization and hardware loop counters on the speculation and overall execution time. Our proposal is compatible with a wide range of proposed Spectre countermeasures including software- and hardware-based approaches. Since we reduce the amount of speculation, we inherently accelerate these solutions by minimizing occasions for pipeline stalls and flushes. We implement our proposed design changes in the LLVM compiler framework and evaluate them using a superscalar, out-of-order CPU simulated in gem5. In addition, we use a binary rewriting tool that optimally schedules instructions to reduce the speculation windows. Our results show an average 10% reduction in speculative instructions for our benchmarks and a performance improvement of up to 16.6% on the Spectre-secure CPU implementation.

To summarize, our contributions are:

- We explore different code transformations that reduce the Spectre attack surface without adding performance overhead and depending on the target architecture require zero to little changes in hardware.
- We show how these transformations can be paired with a broad range of Spectre countermeasures to reduce the performance costs induced by the countermeasure.
- We evaluate the performance improvements of our modifications by implementing compiler support in the LLVM framework, a binary-rewriting tool and a proof-of-concept Spectre-secure RISC-V CPU in the gem5 simulator.

The work described in this paper has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and under the Priority Program SPP 2253 Nano Security (Project RAINCOAT - Number: 440059533).

979-8-3503-4859-0/DATE24/© 2024 EDAA

II. Background

A. Transient Execution Attacks

Transient execution attacks have been introduced with Spectre [6] and Meltdown [7]. Meltdown can easily be mitigated in hardware since the attack exploits a microarchitectural bug that occurs on faults. Spectre on the other hand exploits microarchitectural leakage on mispredicted branch instructions. Since branch prediction is an essential part of today's microarchitectures, mitigating this attack without sacrificing performance is generally considered to be a hard problem. A large variety of Spectre attack variants have been proposed. The original Spectre-v1 [6] exploits conditional branch prediction using the Pattern History Table (PHT). Spectre-BTB [6] exploits indirect branch target prediction using the Branch Target Buffer (BTB). With Spectre-RSB [8], attackers exploit mispredictions on the Return Stack Buffer (RSB) while Spectre-STL [5] exploits speculative Store-to-Load forwarding (STL).

In this work, we focus on countermeasures against Spectre-v1, motivated by the lack of mutually secure and efficient countermeasures for this variant. There are different types of Spectre-v1 gadgets, two of which are shown in Listing 1.

Listing 1: Spectre-v1 gadgets

```
1 if(x<n) // Spectre-v1 index gadget
2     y = a2[a1[x]];
3
4 if(x<n){ // Spectre-v1 compare gadget
5       if(a1[x] == c){ ... }}
```

In case of the index-gadget, the branch prediction for the if-condition (Line 1) is miss-trained by the attacker to access memory out of the bounds of array `a1` and leak the secret value via the cache hierarchy by using it as an index for array `a2` (Line 2). This kind of gadget is very powerful, if `x` is completely controlled by the attacker, because it enables them to access and leak any value in memory. At the same time, this gadget is an artificial one that is hard to find in real-world code [2].

However, there are more ordinary code snippets that can be used for a transient execution attack that have a much higher prevalence [2]. The compare gadget shown in Line 4 of Listing 1 is sufficient to leak whether the secret value located at `a1[x]` equals `c`.

B. Countermeasures against Transient Execution Attacks

Many countermeasures against Spectre attacks have been proposed, some solely based on software transformation, others on hardware modification, and some requiring changes in both domains. There are multiple stages and parts of a Spectre attack where a countermeasure can intervene:

Speculation: The first point where Spectre countermeasures can be applied is the (incorrect) speculation. One of the earliest proposed software defenses use `fence` instructions to enforce serialization on potential Spectre gadgets [1]. BasicBlocker [12] represents another (costly) approach, where speculation is entirely eliminated from the CPU.

Loading Secrets: Based on the observation that speculation, even when incorrect, is not an inherent problem as long as it can not be used to access secret data, another kind of countermeasure can be realized. For example, ConTExT [10] requires the annotation of secret data in the code to block speculative memory accesses to this data at runtime.

Leaking values: One step further in a Spectre attack is the transmission of the (potential) secret data over a covert channel. For example, SLH [3] masks loaded values after a misprediction to only leak a fixed value. STT [14] implements dynamic taint tracking of speculatively loaded values and interrupts the execution before reaching a leaking instruction.

Covert Channel: These type of countermeasures aim to close a specific covert channel, that is used to transmit the secret data from the victim process to the attacker as the last step of a Spectre attack. For example, InvisiSpec [13] targets the cache.

III. Concept

Spectre attacks can use arbitrary covert channels which makes it difficult to protect against it by mitigating the side channels themselves. Thus, a Spectre-immune system is hard to realize with dedicated defences focusing on specific channels. Countermeasures that prevent loading secret values, rely on the programmer to correctly annotate all secrets in the code and need to correctly track them at runtime – which might turn out to be vulnerable to new attack vectors.

We observe that the commonality of countermeasures that neither target a specific covert channel nor specifically prevent the loading of secrets is that they require stopping or delaying the speculative execution of potential Spectre gadgets at some point, thus introducing overhead for these code patterns. Existing countermeasures that either impact the speculation at an early stage or stop the transitive and speculative execution to prevent leaking values are not perfectly precise and conservatively block more patterns than necessary. Therefore, reducing the number of code sections affected by the countermeasures with less overhead than the countermeasure would cause will speed up programs with these Spectre countermeasures in place.

The research question of this paper is how to transform code with small performance overhead to reduce code patterns that induce large overhead in a broad range of Spectre countermeasures. This is partly inspired by Specfuscator [11], a Spectre countermeasures that simply removes branches from the software entirely. While Specfuscator does not result in acceptable performance in general, the overhead can be surprisingly low for some programs and raises the question, if a sanitized use of code transformations in combination with a defence mechanism can lead to better overall results.

In the following, we present three different transformations that increase the performance of Spectre countermeasures by reducing the amount of speculation required. Depending on the target architecture, they do not require changes in the ISA and hardware since they rely on existing instructions. For

the RISC-V ISA as target of our evaluation, however, a few additional instructions are required.

A. Control-Flow Linearization

As the first measure to reduce speculation, we apply partial control-flow linearization (also known as *if-conversion*). Inherently, removing branch instructions from the code eliminates the need for speculation at those occasions and Spectre countermeasures do not need to delay the execution.

Let c be a condition that is computed in block S and evaluates true or false. Based on c, a conditional branch leads the control flow to either block A or B. In block A, variable a is stored in register x, in block B, variable b is stored in x instead. Block A and B finally branch to block E. A linearized version of this code would instead consist of a single basic block where after the computation of c, a branchless expression moves the correct variable into register x, based on c. A single instruction ideally implements this branchless conditional move, often named as `cmov` (conditional move) or `sel` (select). Alternatively, it can be expressed with arithmetic or Boolean instructions, e.g.: $x = (c \wedge a) \vee (\neg c \wedge b)$.

When a dedicated instruction is available, in a simple example like this, linearization can even be faster for a processor that is not restricted in its speculation, particularly when the branch condition is hard to predict [9]. For a scenario like ours, where branching is potentially more expensive, partial linearization becomes an even more useful transformation, also for more complex code patterns.

B. Early Branch Condition Evaluation

The necessity for branch prediction and speculation stems from the fact that a branch condition is rarely evaluated when the processor needs to determine which instructions should be fetched next. Some architectures like MIPS, try to mask the overhead of the branch evaluation by implementing a *branch delay slot*, which allows the processor to execute instructions while waiting for the branch condition to be resolved. However, the number of delayed instructions is limited (e.g., one or two). BasicBlocker [12] proposed rescheduling the branch instruction within a basic block, effectively creating a variable-size branch delay slot at the cost of a major ISA and microarchitectural redesign. However, the techniques above do not provide good performance in the presence of longer superscalar pipelines and advanced branch predictors.

As our second mechanism to reduce the amount of speculation, we propose separating the branch condition from branch instruction itself. With our transformation, the evaluation instruction sets a flag which is used by the branch instruction to take, or not take the branch. If the two instructions follow each other directly, there is no performance benefit. However, the separation allows the two instructions to be scheduled as far apart as the dependencies allow, giving some extra cycles for the branch evaluation.

C. Zero Overhead Loops

The original idea of a zero overhead loop (ZOL) mechanism is to reduce the overhead of instructions for loop management. Instead of computing a counter explicitly in a general-purpose register, this logic is done implicitly with dedicated hardware. Especially for very short loops with a high loop trip count, a ZOL leads to a faster execution time due to fewer instructions executed and better instruction cache efficiency. Loop unrolling, an alternative measure to speed up loops, is usually only done partially or for loops with a low trip count because of its impact on the code size. ZOLs are often implemented for digital signal processing and can be found in multiple ISAs: PowerPC, ARMv8.1-M and Hexagon V5X all feature similar concepts that differ slightly in the setup instructions and the constraints they put on the loops. Nested loops can also be realized with ZOLs when multiple sets of counters are available, as in the Hexagon ISA, which supports two sets of ZOLs.

In general, for a loop to be executed with a ZOL mechanism, the maximum loop trip count must be known at run-time before entering the loop to be able to setup the loop counter. If the loop trip count can be determined at compile time, the loop counter setup can be realized with an immediate value. To support calls in the loop body, the state of the ZOL must be saved. Depending on the implementation, early exits in the loop can be supported. In the context of Spectre countermeasures, we see another advantage of ZOLs. Given a loop that iterates over an array, data is loaded in the body of the loop, and then further operations are done on this data. As a conditional branch controls the execution of the loop body, this could potentially become a Spectre gadget and must therefore be protected by a countermeasure, which comes with a performance overhead. A Spectre-secure CPU could, in this case, stall the pipeline at some point in the loop body. If the loop is executed many times, this substantially contributes to the execution time. With a ZOL, however, no speculation is required on the branch instruction, as the trip count is known.

The ZOL mechanism that we implemented resembles the one of PowerPC. We add one special purpose register to save the state of the counter. The counter value can be initialized with dedicated instructions with an immediate value or loaded from a general-purpose register. We introduce a new branch instruction that implicitly decrements the counter and branches to the beginning of the loop, which is marked by a dedicated instruction, if the counter is not zero.

D. Mitigating Spectre

To evaluate our transformations, we have to pair it with a Spectre countermeasure. We have chosen to implement a speculation barrier in the hardware, that stalls the pipeline when a transient memory load occurs. This makes the processor immune to all Spectre-v1 attacks, where a secret is loaded from memory, and even covers prefetch gadgets. In contrast to approaches like STT [14], this approach requires no assumptions on covert channels which may evolve in the future. This threat model represents the *futuristic* attacker introduced in prior work [13].

IV. Implementation

We implemented a prototype of our concept based on the open RISC-V ISA, utilizing the gem5 simulator, the LLVM compiler framework, and an internal framework for binary rewriting.

A. ISA extension

We introduce a custom RISC-V instruction set extension that adds comparisons syntactically similar to the existing branch instructions. For example, the `cmplt` instruction is directly derived from the `blt` instruction. It includes the registers to be compared, as well as the branch offset. Compare-instructions must always be paired with a *branch notification*, which is implemented as a `branch` instruction. It branches depending on the result of the comparison.

For code linearization we included the `cmov` instruction from the preliminary RISC-V binary manipulation extension. Our ZOL mechanism as described in Section III-C, is based on three new instructions: `loopC` (set), `loopB` (begin) and `loopE` (end) and two special purpose registers. In particular the `loopC` and `loopB` instructions can alternatively be realized with the existing `csrw` instructions in RISC-V. The loop counter register can either be set with an immediate value, when the loop trip count is known at compile time or by a register value, when the loop trip count is known at compile time before entering the loop. Multiple sets of loop registers and logic would enable nested zero-overhead loops. For this prototype, we use a single configuration for simplicity.

B. Compiler Modifications

Starting with LLVM 14.0.0 as our baseline, we add support for the `cmov` instruction by cherry-picking from the relevant compiler parts that implement support for the RISC-V B-Extension. This already enables the compiler to perform linearization transformations on small code fragments via pattern matching. Additionally, we utilize the generic *SimplifyCFG* pass for further linearization. How much and for which code patterns linearization is applied can be controlled with multiple parameters in the compiler. Its performance improvement or degradation highly depends on the CPU and will also vary depending on the Spectre countermeasures. To not optimize specifically for our implementation target, we use conservative and default parameters, which we validated by measuring the performance of our transformations and extensions without Spectre countermeasures.

We implemented assembler support for ZOL instructions but leave additional optimizations for the automatic selection and code transformation for future work. Instead, we chose to insert ZOL instructions manually.

C. Binary Rewriting

For our proof-of-concept, we utilize a combination of compiler passes as well as binary rewriting to schedule the compare instruction as early as possible within each basic block ending with a branch statement. First, an LLVM pass identifies suitable branches and inserts a placeholder directly above each branch. The compiled binary is then lifted into an intermediate language, which allows for the extraction of instruction semantics as well as basic block boundaries, which are required as the comparison should not be moved outside of its original basic block. For each identified placeholder, we insert the appropriate comparison and overwrite the original branch instruction with our new branch notification instruction. A def-use chain is employed to identify whether moving the compare instruction up is possible. For better results, we optionally reorder the instructions within a basic block in order to create a more suitable instruction sequence, which allows to move the compare instruction further up. We compare the results of the original def-use chain and the reordered instruction sequence and pick the one with the highest result, and move the corresponding bytes within the binary. We want to note that this approach as well as improved rescheduling strategies can also be implemented into the compiler in future work.

D. CPU Modifications

We first describe the modifications required to support the new instructions and hardware features. Then, we describe our implementation of a Spectre-secure speculation mechanism.

1) New Instructions: We implemented support for the new instructions to the microarchitecture simulator gem5 for RISC-V. In particular, we added 5 new comparison instructions (`cmpeq`, `cmpne`, `cmplt`, `cmpge`, `cmpltu` and `cmpgeu`), the new branch notification instruction (`branch`), the loop-counter instructions (`loopC`, `loopB` and `loopE`), and the predicative instruction `cmov`. Notably, our modified CPU implementation still supports all traditional RISC-V branches allowing for backwards compatibility to binaries that have not been adapted.

For `cmov`, we extended the register definition of the RISC-V ISA implementation by a new source register *RS3* according to the draft *bitmanip* specification version 0.92. The compare-, branch-, and loop instructions require changes to the CPU directly. We chose gem5's O3 CPU model, which simulates a superscalar, out-of-order CPU with six pipeline stages. The fetch and decode stages thereby represent the CPU frontend, which implements the main functionality of the new compare / branch instructions. Unlike traditional branch instructions of RISC-V, the compare sets an internal CPU register to the branch target instead of modifying the PC directly. The CPU tracks new compare instructions in the frontend and matches them to the following `branch` instruction. In semantically correct programs, every compare is followed by an arbitrary number of non-control-flow instructions and then finally a `branch`. Hence, the CPU only needs to keep track of one comparison at a time. Note that it may happen that a compare or branch instruction is fetched to the CPU frontend but later flushed due to a branch misprediction or a memory violation. Then, the comparison or branch must also be flushed from the tracker in the frontend and the mapping of compare and branch instruction must be reverted. When a `branch` instruction reaches the frontend, the CPU predicts a control-

flow change to the address stored in the branch target register. This may either be the result of the branch prediction (if the comparison is not yet resolved), or the correct branch target (if the comparison was resolved). Only if the comparison is executed and the CPU finds that the branch prediction was incorrect, the pipeline needs to be flushed and the execution must continue at the `branch` (notably, not on the actually mispredicted compare).

The loop counter mechanism is implemented in a similar fashion. The `loopC` instruction sets an internal loop counter value register. The frontend tracks incoming `loopC` instructions until they are retired. The CPU must only be able to track one `loopC` instruction since any following instruction would overwrite the result. The instruction address is stored in an internal loop-begin register by the `loopB` instruction. Since we require hardware-assisted loops to perform at least one iteration, the CPU can continue execution until a `loopE` instruction. As long as the `loopC` is not executed, the frontend is blocking when a `loopE` instruction is found. The loop counter is decremented on every `loopE` instruction and sets the PC to the loop-begin address as long as the loop counter is not zero. Otherwise, the `loopE` acts as a `nop`.

2) Spectre-Aware Speculation: To evaluate our modifications in a setting where speculation is restricted to protect against Spectre attacks, we implement a speculation barrier in gem5. Therefore, the CPU maintains a list of unresolved comparisons and (traditional) branches. This is typically done int the ROB. When a memory referencing instruction occurs, the CPU blocks the execution of all following instructions until the comparison is resolved. This protects against Spectre attacks since the CPU cannot encode information from memory to the microarchitectural state in a speculative context.

V. Evaluation

In the following, we evaluate the applicability and performance of our concept. Similar to [12] we chose the *embench* benchmark suite [4] to evaluate our proposed instruction extension. Its compactness is a major advantage, as we insert the ZOLs manually. It also speeds up the binary analysis post-compilation. The results of the transformation are shown in Table I.

First, we evaluate the performance of the instrumented benchmarks on the baseline CPU that has no countermeasure against Spectre attacks and thus no limitation and overhead during speculation. On average, our modifications lead to a speed-up of 1.5%. This improvement can be mainly assigned to the code linearization using the `cmov` instruction. Experiments show comparable performance advantages of the `cmov` instruction on a x86 Skylake CPU [9].

A. Speculative Code Execution

We now investigate how our modifications affect the number of speculative instructions in the CPU frontend. We define an instruction as speculative if it is processed in the CPU frontend when the result of previous branch instructions is not yet known. Note that speculative instructions are not necessarily rolled back. If the branch prediction is correct, they will be executed normally. However, even though the branch predictor in most cases *is* correct, even an almost-perfect branch predictor could not prevent speculative execution attacks. That is since the attacker maliciously manipulates the prediction mechanism to force a misprediction. Hence, our aim is to reduce the speculation generally and not just improve the branch prediction. Thus, we count the instructions that *could be* rolled back instead of counting only those, that are. The results are listed in Table I (right column) and show a significant reduction in speculative instructions for some benchmarks (e.g., for *crc32*, the number of speculative instructions is halved) and almost no effect for others (e.g., the *cubic* benchmark). For the *aha-mont64* benchmark, the percentage of speculative instructions is reduced from 97% to 79% by our modifications. Our toolchain replaced 22 conditional branches with conditional move instructions and one ZOL during transformation. The rescheduling of compare instructions was able to only place on average 0.8 instructions between compare and branch. Hence, the reduction of speculative instrucitons in *aha-mont64* mostly stems from the lineraization. For *cubic*, neither linearization could be applied nor a ZOL. On average, 3.5 instructions could be placed between the branch condition and the actual branch. However, this alone was not sufficient to significantly reduce the number of speculative instructions in this example. Similarly, the *crc32* and *edn* benchmarks did not offer many possibilities for linearization and ZOLs. Here, however, more instructions could be placed between the compare- and branch instructions which led to a reduction of speculative instructions by about 25% for both benchmarks.

The *matmult-int*, *nettle-sha* and *nettle-aes* benchmarks speculate the least of all. That is since the C code is already optimized for linearization and contains very few conditional branches. This leads to large basic blocks with almost linear code execution and therefore, vast opportunities for rescheduling of compare instructions. Similarly interesting is the *nsichneu* benchmark in which almost all instructions were processed

TABLE I: Conditional branch- (CBR), Transformation-, and speculatively executed instruction statistics of the benchmarks.

Program	CBR before	CBR after	`cmov`	ZOL	Avg. Shift	Spec. before	Spec. after
aha-mont64	32	10	18	1	0.8	97%	79%
crc32	3	2	0	1	5.7	58%	27%
cubic	7	7	0	0	3.5	82%	82%
edn	22	22	0	0	6.8	82%	59%
huffbench	56	44	9	0	2.5	95%	93%
matmult-int	5	4	0	1	98.6	18%	5%
minver	44	40	1	4	2.7	94%	81%
nbody	44	42	0	2	1.7	88%	89%
nettle-aes	36	25	8	5	21.4	21%	6%
nettle-sha256	22	20	0	2	2.4	7%	6%
nsichneu	641	641	0	0	0.0	100%	99%
picojpeg	705	384	272	0	1.4	95%	75%
qrduino	316	179	210	3	1.0	93%	86%
sglib-comb.	492	481	9	2	0.8	94%	90%
slre	143	122	24	0	1.2	89%	80%
st	14	14	0	0	5.5	96%	85%
statemate	139	135	0	0	0.6	59%	41%
ud	13	13	0	0	6.6	90%	82%

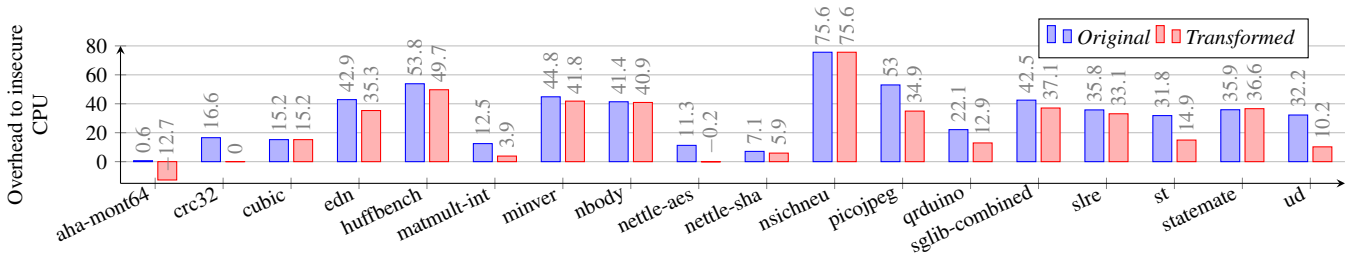

Fig. 1: Performance overhead of the original- and transformed benchmarks on a Spectre-secure CPU compared to the execution of the original benchmarks on the Spectre-vulnerable CPU.

speculatively. Our compiler framework did not utilize any possibilities for linearization and we could not instantiate hardware assisted loops. Moreover, only very few compares could be rescheduled, resulting in a negligible reduction of speculative instructions.

B. Performance on the Spectre-Secure Core

To analyze the impact for a Spectre-secure processor.´, we compare the performance of the original benchmarks with the transformed benchmarks in Figure 1. For the original code, the performance overhead is up to 75%. While the upper bound for the transformed benchmarks is similar, looking at the individual benchmarks shows a significant performance improvement for most benchmarks. For the *aha-mont64* benchmark the Spectre-secure CPU even outperforms the insecure CPU with our modifications. Earlier, we have identified *crc32* as a benchmark where the number of speculative instructions has been significantly reduced by the modifications. The effect of this reduction is clearly visible in the benchmark results of the Spectre-aware CPU. We were able to compensate for the entire overhead that occurred by implementing the speculation barrier. Similar results occur for *matmult-int* and *nettle-aes*.

As before, there are also cases where our modifications could not be applied sufficiently to make a difference in the runtime on the Spectre-aware CPU. For example, the performance of the *cubic* and *nsichneu* benchmarks was not affected at all. There was little opportunity to make use of our modifications which explains these results. Moreover, the *nsichneu* benchmark heavily relies on speculation.

On average, our modifications reduced the performance overhead resulting from the Spectre-countermeasure from 32% to 24%. Naturally, these numbers will vary for different countermeasures and different CPUs. For example, if the base overhead of the countermeasure is reduced, the performance gained will also be smaller. However, the results clearly show that reducing speculation in the CPU microarchitecture does help to reduce the overhead. With attacks constantly evolving, we expect future countermeasures to require stricter limitations for speculation, hence increasing the effect of the proposed code modifications.

VI. Conclusion and Future Work

In this paper, we have shown that minor code transformations can have a significant performance impact on Spectre countermeasures. By reducing the number of speculatively-executed instructions, we are able to accelerate a broad range of software and hardware countermeasures that have to restrict speculative execution at some point for security. We believe that a tight integration of our proposed changes into the compiler will further increase the effectiveness of this approach.

References

[1] AMD. Software techniques for managing speculation on amd processors. https://www.amd.com/system/files/documents/software-techniques-for-managing-speculation.pdf, 2018. Accessed: 2023-03-17.

[2] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX*, pages 249–266, 2019.

[3] Chandler Carruth. Speculative load hardening – a spectre variant #1 mitigation technique. https://llvm.org/docs/SpeculativeLoadHardening.html, 2018. Accessed: 2022-11-01.

[4] Free and Open Source Silicon Foundation. Embench: A modern embedded benchmark suite, 2023. Last accessed: 23.03.2023.

[5] Jann Horn. speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018. Accessed: 2023-03-17.

[6] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. volume 63, pages 93–101. ACM New York, NY, USA, 2020.

[7] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, et al. Meltdown: Reading kernel memory from user space. volume 63, pages 46–56. ACM New York, NY, USA, 2020.

[8] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *SIGSAC*, pages 2109–2122, 2018.

[9] Marcin Osowski. Comparing the performance of intel's cmov instruction with a conditional branch + mov pair. https://github.com/marcin-osowski/cmov, 2018. Accessed: 2023-03-17.

[10] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. Context: A generic approach for mitigating spectre. In *NDSS*, 2020.

[11] Martin Schwarzl, Claudio Canella, Daniel Gruss, and Michael Schwarz. Specfuscator: Evaluating branch removal as a spectre mitigation. In *Financial Cryptography and Data Security*, pages 293–310. Springer, 2021.

[12] Jan Philipp Thoma, Jakob Feldtkeller, Markus Krausz, Tim Güneysu, and Daniel J Bernstein. Basicblocker: Isa redesign to make spectre-immune cpus faster. In *RAID*, pages 103–118, 2021.

[13] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy (corrigendum). In *MICRO*, pages 1076–1076, 2019.

[14] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data. *IEEE Micro*, 40(3):81–90, 2020.