Secure Speculative Execution via RISC-V Open Hardware Design

Majid Sabbagh
sabbagh.m@northeastern.edu
Northeastern University
Boston, MA, USA

Yunsi Fei
yfei@ece.neu.edu,
Northeastern University
Boston, MA, USA

David Kaeli
kaeli@ece.neu.edu
Northeastern University
Boston, MA, USA

ABSTRACT

Recently, there has been a sharp rise in the number of microarchitectural vulnerabilities being exploited on modern processors, including a variety of Spectre attacks. Current software mitigation has significance performance overhead and often requires modification of the source code. Architecture augmentations are mostly verified by simulators, without real hardware implementation and evaluations.

In this project, we propose an efficient taint-tracking solution for secure speculative execution (SSE-RV) that protects against the most prominent speculative execution attacks. We take a secure-by-design approach, leveraging the RISC-V open hardware ecosystem, and implement our taint tracking mechanism in the latest Berkeley Out-of-Order Machine (SonicBOOM). We prototype our SSE-RV processor on an FPGA running Linux. Our results show that we can protect against Spectre-v1, v2, and v5, while achieving 0.66 CoreMark/MHz for single small BOOM core performance. We also synthesize our processor core targeting 130nm BiCMOS technology for implementation cost evaluation. Our design only introduces a 0.42% increase in the gate count and a 1.7% increase in the total core power consumption, compared to an unprotected core. Our defense scheme is general and can be extended to protect against other transient execution attacks.

CCS CONCEPTS

• **Security and privacy** → *Side-channel analysis and countermeasures*; **Hardware attacks and countermeasures**.

KEYWORDS

Spectre attacks, taint tracking, memory fencing, RISC-V processors

ACM Reference Format:
Majid Sabbagh, Yunsi Fei, and David Kaeli. 2018. Secure Speculative Execution via RISC-V Open Hardware Design. In *CARRV'21: Fifth Workshop on Computer Architecture Research with RISC-V, June 17, 2021, Virtual.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Modern processors incorporate many performance features that are designed to leverage the spatial and temporal parallelism present in the instruction streams. Out-of-order (OOO) and speculative execution have been used to boost performance of processors. Although these microarchitectural optimizations are critical to improve the instructions executed per cycle (IPC), they can inadvertently present microarchitectural security vulnerabilities. Meltdown [15] and Spectre [13], and a series of transient execution attacks are the result of these security-blind hardware optimizations. Transient execution attacks exploit microarchitectural vulnerabilities to access sensitive data illegally (bypassing security checking in place), and leak the data to an adversary application through a microarchitectual covert channel. The most commonly used covert channels rely on data or instruction *caches*, while more recent ones target translation lookaside buffers (TLBs), branch target buffers (BTBs), pattern history table (PHT), single-instruction multiple-data (SIMD) units, execution ports, and floating point units [2, 6, 7, 9, 13, 14, 18].

Various schemes have been proposed to protect against Spectre attacks at different levels of the compute stack. For software approaches, memory barriers are added after the speculative constructs at an application level, [5], eager floating point unit-FPU switching is performed at an operating system (OS) level [20], and the branch prediction can be disabled at a firmware level [19]. For hardware approaches, existing modules can be augmented as done in a dynamically-allocated way guard-DAWG [12], or new secure modules can be introduced, e.g., an invisible speculative buffer [25] and speculative taint tracking-STT [28]. Other approaches such as ConText [17] require modifications across application, compiler, OS, and hardware.

However, there are three common weaknesses present in current protection mechanisms. First, there is a high degree of implementation overhead in execution time, area, or power consumption. For example, adding fence instructions after each branch to make them non-speculative can impact the performance by 88% [5]. Disabling speculation [19] at firmware level can lead to 7x performance loss or even more. Hardware solutions like on-chip shadow buffers come with huge area and power overheads, e.g., InvisiSpec[25] incurs $35{,}000\mu m^2$ additional area usage and extra $8.8pJ$ of dynamic energy per read/write transaction under $16nm$ technology. Second, some require extra programming effort. For example, ConText [17] require the user to annotate sensitive data in the source code or do code transformation, and also compile the code with specific flags. Third, there lacks real hardware prototyping. Majority of the hardware approaches are only evaluated by simulation, including STT [28], Speculative Data-Oblivious Execution [27].

In review of the existing approaches, we recognize hardware-level taint tracking [28] appears to be the most effective and efficient solution against transient execution attacks. In this paper, we take a secure-by-design approach and leverage the RISC-V open hardware ecosystem to design an efficient taint tracking scheme to guard against Spectre attacks, SSE-RV. We target the latest generation of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CARRV'21, June 17, 2021, Virtual
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

Berkeley Out-of-Order Machine (SonicBOOM) [29]. The Spectre attacks include Spectre-v1 (bypassing bounds checking), Spectre-v2 (branch target injection), Spectre-v5 (return stack buffer attack). The reason for picking these three is they involve different microarchitectures but share similar attack procedure and consequence. We define the following design principles for our protection scheme: 1) reuse the existing microarchitecture as much as possible to reduce area and power overhead, 2) design simple and easily expandable hardware, 3) minimize the performance penalty by selectively blocking the transient instructions and stalling the pipeline, while fully benefiting from OOO and speculative execution.

We prototype the SSE-RV protected SonicBOOM on a digilent Genesys-2 FPGA board, running a 64-bit Debian OS. We port the open-source Proof-of-Concept (PoC) codes for the three selected attacks [8] to SonicBOOM and evaluate our prototype. Our results show protection against all three variants leaking none of the secret byes, while the original SonicBOOM is vulnerable. We synthesized our SSE-RV core targeting a 130nm BiCMOS ASIC technology and found an increase of only 0.42% in the gate count and 1.7% increase in the total core power consumption, compared to the original core.

The contributions of this research are summarized as follows:

- We implemented a novel taint tracking architecture based on SonicBOOM, SSE-RV, that can protect against Spectre attacks, outperforming the state-of-the-art.
- We developed an FPGA prototype for the proposed SSE-RV.
- We evaluated the security guarantees delivered by our protection scheme, as well as area/power/performance overheads.

The rest of this paper is organized as follows. In Section 2, we present background materials. In Section 4, we describe our experimental setup, and present our security and implementation evaluation results. In Section 5, we discuss the limitations of our work and propose future extensions. Finally, we conclude the paper in Section 6.

2 BACKGROUND

Next, we review out-of-order (OOO) execution and speculative execution on a CPU, particularly vulnerable microarchitectural features on SonicBOOM architecture. We describe the different phases of a general transient execution attack, and take the Spectre variants for example. We also compare state-of-the-art taint tracking mechanisms against Spectre attacks.

2.1 SonicBOOM Vulnerable Speculative Execution

The SonicBOOM microarchitecture [29], just as counterparts on other modern processors, has two performance features that become problematic in terms of security: OOO execution and speculative execution. In OOO mode, instructions are scheduled to available functional units for execution in parallel, in a different order than the program order [11]. A reorder buffer (ROB) tracks the status of each instruction in the pipeline. Instructions are queued in the ROB in instruction fetch order, and committed when their execution is complete and they reach the head of the ROB.

Speculative execution addresses the penalty incurred by control flow instructions. With the help of some prediction microarchitecture, instructions are executed speculatively in an aggressive fashion, rather than being stalled until the condition is resolved. If a speculative instruction turns out to be on the correct path, it will be committed and become architecturally visible; otherwise, it (and the following transient instructions) will be squashed, and the processor state is rolled back to a valid state.

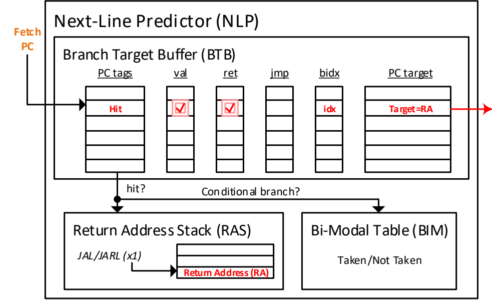

Figure 1: SonicBOOM next-line predictor (NLP).

The RISC-V control flow instructions include conditional branches (e.g., *branch if less than or equal*, `ble`), unconditional jumps (e.g., *jump register*, `jr rs`, or *jump and link*, `jal offset`), call instructions (`call offset`), and return instructions (`ret`). These instructions are composed of fused μops. For example, a `call` instruction is implemented by a μop for address generation followed by a `jalr` μop to store the return address and jump to the subroutine start address. Also, a `ret` instruction is implemented using a `jalr` μop, with register `x1` (`ra`) containing the return address.

In SonicBOOM, all control flow instructions may trigger speculative execution, supported by microarchitecture prediction units. There are two levels of instruction flow prediction in SonicBOOM: 1) a simple *next-line predictor* (NLP) and 2) a more sophisticated multi-way *backing predictor* (BPD). There are five sub-stages (F0-F4 cycles) in the SonicBOOM's Fetch stage. During the Fetch stage, a PC address is used to look up the instruction TLB for the physical address. I-Cache is then looked up for a fetch packet (instruction block). If there are control flow instructions in the fetch packet, they can be detected and used to consult NLP and BPD for possible control flow change (redirect logic to determine the target address of control flow instructions).

Fig. 1, shows the structure of the NLP in SonicBOOM, which consists of a branch target buffer (BTB), a Return Address Stack (RAS), and a Bi-Modal (counter) Table (BIM). The BTB is fully-associative and the RAS is a LIFO structure with a limited number of entries (16-32). The goal is to make a fast, but reasonably accurate, target address prediction. The PC address's tag is used to probe the BTB to find a match. If there is a BTB hit, a control flow instruction exists in the fetch packet, and one of the three buffers is consulted for prediction depending on the instruction. If it is a return instruction, RAS is looked up. If it is an unconditional jump instruction, the BTB directly predicts the target. If it is a conditional branch, the hysteresis bits in the BIM are used to determine whether the branch is taken or not taken. The target address prediction is only needed when predicted taken, and is provided by the BTB. The BTB learns from the target history of taken branches or jumps. The RAS is updated once the instructions in the Fetch Packet have been decoded: return address is pushed onto the RAS when the instruction is a call; the top of RAS is popped when the instruction is a return.

The BIM hysteresis bits cannot learn very complicated or long history patterns. SonicBOOM uses a BPD, a high performance

TAGE[21] predictor that makes direction predictions with a high accuracy for dense areas with many conditional branches. The BPD is accessed throughout the Fetch stages and in parallel with I-cache and NLP accesses. It eventually provides a bit-vector of taken/not-taken predictions, where each bit corresponds to an instruction in the fetch packet (1: taken, 0: not taken). The BPD is generally updated during the *commit* stage.

Similar to other processors with branch prediction, SonicBOOM *speculatively* executes the predicted target addresses of a predicted **taken** conditional branch, a jump, or a return instruction. This transient execution changes the microarchitectural state of the processor temporary. When the condition is resolved or the real target is available, in the case of a misprediction, the transiently executed instructions will be squashed and the processor state (including the PC) will be rolled back. Note that the branch speculation may resolve as early as in one fetch sub-stage, or as late as a few hundred cycles later in case of a cache miss for a data/instruction on which the branch is dependent. Therefore the *speculation window* ranges from a couple of cycles to hundreds of cycles. The length of this window is critical in the success of a Spectre attack. An adversary can exploit transient execution in the speculation window to tap "architecturally" non-accessible data and transmit it through a covert channel (e.g., the L1-Cache).

The Load/Store Unit (LSU) is responsible for deciding when and how to *issue* memory operations to the memory subsystem. Decoded memory instructions generate `uopLD`, `uopSTA` or `uopSTD` *μop*s, reserve entries in corresponding queues in the LSU. These *μ*ops are issued to the address generation unit (AGU) (`memaddrcalc`) for calculating the virtual addresses which are sent to TLB. If there is a TLB hit, the physical address is written into the (load/store) queue entry in the LSU and the access request is sent to the data cache (D-cache). Otherwise there is no physical address available for data cache access. It takes three cycles for D-cache to allocate any misses in the miss status holding registers (MSHRs) and request data from L2 cache, eventually used in the WriteBack (WB) stage. There is an extra overhead of eight cycles for every two loads with the two MSHRs. Loads are *optimistically* fired to memory upon arrival in the LSU to benefit from OOO execution.

2.2 Spectre attacks

Transient execution attacks, including Spectre attacks, can be described in six major phases [3]. These phases are:

1. **Preface**: the attacker establishes the desired conditions for triggering the transient execution. For example, by (mis)training a branch predictor unit to a certain direction for a Spectre-v1, the attack can provide an invalid input (out-of-bounds value) to the trigger instruction. Also the receiver of the covert channel can be set by evicting a *probe* array out of the cache.
2. **Trigger**: the attacker will trigger transient execution (in data or control flow) through an instruction (e.g., a branch instruction in Spectre-v1).
3. **Tap**: a sequence of transient instructions tap on the secret data and load it to a speculative register (program invisible).
4. **Covert channel**: a transmitter instruction (e.g., a speculative load) will encode the secret value and transmit it over a covert channel (e.g., the data cache).
5. **Administer**: the processor will reset the pipeline by squashing the mispredicted instructions and rolling back to the pre-transient state. An exception may be raised by the processor at this point (e.g., in Meltdown).
6. **Reconstruct**: finally, the attacker will decode the secret received through the covert channel (e.g., access the *probe* array and time each access using the Flush+Reload method [26]).

Spectre attacks is a type of powerful transient attacks which exploit speculative execution. The three Spectre variants we target in this work differ in the first two phases - *preface* and *trigger* and involve different microarchitectures, while the latter four phases are pretty common. Table 1 describes the two phases of the three Spectre variants, including different instructions and microarchitecture, and the transiently executed attack agents.

2.3 State of the art defenses

The defenses against transient execution attacks can target any (combination) of the six attack phases by trading off generalizability and effectiveness. We discuss two state-of-the-art countermeasures against speculative attacks, Speculative Taint Tracking (STT) [28] and Context-Sensitive Fencing (CSF) [22], and show how our SSE-RV approach differs from these schemes.

STT[28] is a hardware-level taint tracking scheme. It taints data that is accessed during a speculative window and *delays* any subsequent *load* with tainted operands until they become untainted. This effectively cuts off the connection between Phase 3 and Phase 4. Untainting of registers happens once the instruction that invoked it becomes non-speculative. This selective speculative execution property makes STT more efficient in performance, compared to solutions which delay the execution of *every* load instruction until they reach the "visibility point" (e.g., when all older control flow instructions have resolved) as in InvisiSpec[25]. STT adds 14.5% performance overhead relative to an unprotected machine.

CSF[22] is a micro-code level defense against Spectre attacks. It leverages the ability to dynamically alter the decoding of the instruction stream, injecting fences only when dynamic conditions indicate they are needed. CSF uses a decoder-level dynamic information flow tracker (DLIFT) to taint the PCs of memory loads early in the pipeline. CSF enables the pipeline to identify tainted accesses before each stage, supporting speculative execution and insertion of fences after loads. CSF is not applicable to processors which do not support microcode updates, such as SonicBOOM. Compared to STT, their scheme has higher performance overhead.

Our approach combines the benefits of STT and CSF, introducing a new effective hardware-level taint tracking protection mechanism with dynamic data memory fencing against Spectre attacks. SSE-RV has a fully functional FPGA prototype, while STT and CSF are just modeled using the gem5 simulator [16]. Compared to STT, SSE-RV does not add extra logic for calculating the "root of taint" in the rename tables, instead it leverages the existing ROB pointers for tracking speculations. Also, SSE-RV does not require adding new fencing *μop*s as in CSF. SSE-RV reduces area and power overheads to less than 0.5% and 1.7%, respectively. While SSE-RV is general, it can make *all types of speculations* which are trackable by ROB secure with minimal changes to the existing hardware.

	Preface	Trigger		
		Instruction	Vulnerability	Attack agent
Spectre-v1	mis-train conditional branch direction prediction	conditional branch	out-of-bound offset	mis-speculated path
Spectre-v2	mis-train BTB for branch target prediction	call or jump	mis-speculated target	malicious function
Spectre-v5	craft a malicious gadget after a call site	return	RAS miss-match with software stack	gadget

Table 1: Elaboration of the two phases for the three Spectre variants

3 OUR PROPOSED APPROACH - SSE-RV

In SSE-RV, the destination register of a *speculative* load instruction is automatically marked as tainted in a hardware taint file. The taint is a metadata of the register, and propagates through the pipeline in parallel to the data. When a subsequent speculative load uses a *tainted address*, sent to the LSU, the enforcement policy blocks the execution of the load (cache covert channel transmitter) by activating the data memory fencing `fence_dmem`. Due to memory fencing, the *speculative data cache refill* is temporarily disabled and the MSHRs is cleared, until the load is no longer speculative. For speculative loads, the speculative condition (trigger) should include conditional branch, unconditional jumps, and returns, to cover the three Spectre variant test cases.

We propose a generic solution using Point of No Return (PNR) signaling from ROB which covers all types of speculation in SonicBOOM.

3.1 PNR in Taint Initialization and Enforcement

The BOOM co-processors (RoCC) expects their instructions to be in-order and do not withhold misspeculation. To enable issuing non-speculative instructions to RoCC, the original BOOM developers add point of no return (PNR) mechanism in the ROB. The PNR index points to "younger" instructions than the ROB commit head, marking the next instruction which might misspeculate (e.g., unresolved branch instruction) or generate an exception (e.g., divide by zero). Any instructions older than PNR are *guaranteed* to eventually commit, although they may have not been executed yet. We use the "fast PNR" scheme, where the next PNR instruction in the ROB can be any number of entries away from the current PNR, not just the next one.

We leverage PNR to guide the SSE-RV taint initialization and enforcement w.r.t. the *speculation condition*. As shown in Fig. 2, the taint initialization unit takes the PNR index (`pnr_idx`), the ROB head index (`head_idx`), and information about the current μop, including its type, the ROB index (`uop.rob_idx`), and the propagated taint bits (`prop_taint`), as inputs. Eq.(1) determines in every cycle if a μop is possible to squash or guaranteed to commit. In SonicBOOM, the ROB entries are not used in monotonically increasing order. Instead, the indexes move between the ROB entries dynamically. To identify if an instruction is older than the PNR, we need to evaluate the *XOR*ed comparison of the μop ROB index, the PNR index, and the ROB head index. For example, one scenario is if $uop.rob_idx < pnr_idx \; AND \; uop.rob_idx >= head_idx \; AND \; pnr_idx >= head_idx$, the μop is "older" than PNR (and between PNR and ROB head) and is guaranteed to commit, otherwise it is still squashable. There are three other scenarios covered by Eq.(1).

$$\begin{aligned}&(uop.rob_idx < pnr_idx)\oplus\\&(uop.rob_idx < head_idx)\oplus\\&(pnr_idx < head_idx)\end{aligned} \tag{1}$$

If the current μop is a load (`uopLD`) or its destination register is previously tainted, and Eq.(1) is <u>not</u> held the corresponding taint bit is set in the taint file (TF). If Eq.(1) holds, it indicates that the current instruction will eventually commit. Eq.(1) covers speculation from both the BPD and NLP (BTB and RAS) where Spectre-v1, and Spectre-v2 and v5 are exploiting, respectively.

3.2 Defense architecture

Fig. 2 shows an architectural overview of SSE-RV. Our speculative taint tracking protection has the following phases:

(1) **Initialization**: A taint file is added as a shadow structure for the physical register file (PRF), which all start at zero.[1]. At the WB stage, if the taint initialization unit decides a register needs to be tainted (with the condition explained in previous section), the taint file entry corresponding to the load instruction is *set to 1*, in addition to data loaded to the physical register. Note, the same address is used to access both the taint file and the physical register file. While a 1-bit taint value is used in this work, our hardware implementation supports any number of bits for the taint values.

(2) **Propagation**: in parallel to each functional unit, there is a taint propagation block with an `OR` logic, with two or three inputs to take in the taint values of the functional unit's operands. The output of the `OR` gate is placed in a 32-entry FIFO queue (32 cycles is an upper bound delay for any of the functional units) and read along with the response of the functional unit. Thus the taint bits propagate synchronously with the data in the functional unit (same #cycles). Finally, the propagated taint output is written to the taint file in the WB stage or used to enforce the protection in the next step.

(3) **Enforcement**: memory loads that use tainted address registers are detected at the AGU (`memaddrcalc`) output. If Eq.(1) does not hold for the load instruction (it is squashable), a `spec_cond` signal is activated. When `spec_cond` is active and the load has a tainted address, a speculative load block signal `spec_ld_block` notifies the LSU to enable the data memory fencing, `fence_dmem`, for all the loads present in the Load Queue. Data memory fencing prevents refilling the D-Cache on misses and clears the MSHRs. The `spec_ld_block` also notifies the front-end hazard control unit to introduce dispatch hazards, forcing the pipeline to stall. This makes the pipeline operate in-order temporarily, until the speculation condition is resolved.

(4) **Untainting**: an untainting step is triggered when the transient instruction passes the PNR (commitable). In the case of a misprediction, the data and the corresponding taint bits become invalid (squashed). In case of a correct prediction, the data is left intact, but all the taint bits in the taint file will be reset to 0 on the falling edge of `spec_ld_block`.

We add a custom Control and Status Register (`CSR`) bit to SonicBOOM for disabling SSE-RV at runtime. By setting the 5th bit in this custom `CSR` we can mask out the `spec_ld_block`, and hence

[1] SonicBOOM's physical register file contains both speculative and program registers.

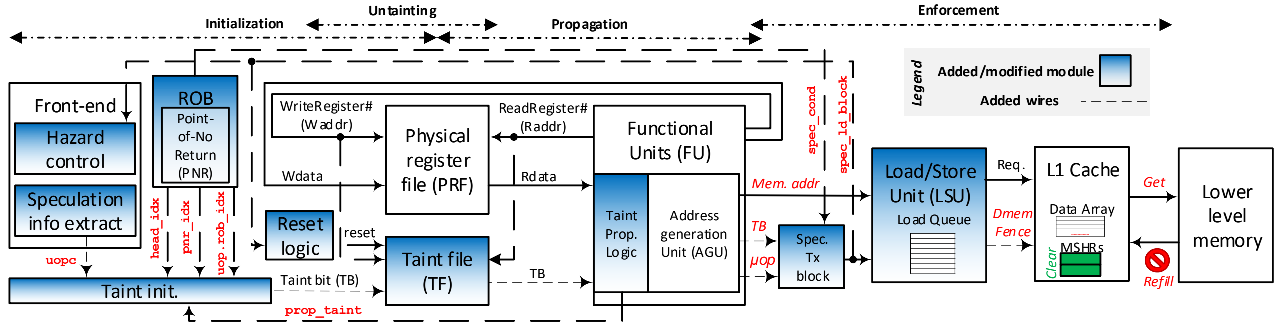

Figure 2: SSE-RV architectural overview.

disable the protection scheme. We use a `csrwi` instruction for updating the CSR. This CSR is used for switching between secure mode and high performance mode for the processor.

4 EXPERIMENTAL RESULTS AND ANALYSIS

In this section we present our experimental results. We describe our experimental setup and the SSE-RV implementation on FPGA. We demonstrate how our approach effectively protects SonicBOOM CPU against Spectre attacks. We also evaluate the performance/power/area overheads relative to an unprotected SonicBOOM CPU.

4.1 Experimental Setup

We use Chipyard v1.4.0 [1] to generate RTL for a single small-core SonicBOOM processor, and use Verilator v4.034 for RTL simulations. Our prototype platform is a Genesys-2 board with a Xilinx XC7K325T-2FFG900C Kintex-7 FPGA using Vivado v2020.1 [23]. The core frequency is set at 100MHz. All of our hardware performance results are collected when running the Debian Linux 64-bit OS with the kernel v5.11.16.

The SonicBOOM small core configuration has a 4-wide fetch, 1-wide decode, 3-wide issue out-of-order speculative pipeline. It has a pipeline for integer operations, and a separate pipeline for floating point operations. Each pipeline has a separate register file (RF). The RAS and ROB each has 32 entries. The LSU connects to 4-way set-associative 16KB DCache and two MSHRs. The I-Cache has the same dimensions as the D-Cache. The L2 cache size is 512KB and the Genesys-2 board has a 1GB DRAM (DDR3).

We run an ASIC synthesis flow for the base core and the SSE-RV core. We target `PnomV1p20T025_STD_CELL_8HP_12T`, a 130nm BiCMOS technology, and use the CADENCE Encounter RTL compiler v14.10 for synthesis, when generating power and area estimation.

We modified the BOOM proof-of-concept codes for Spectre-v1, Spectre-v2, and completed the Spectre-v5 attacks developed by Gonzalez et al.[8] for SonicBOOM. The attacks functionality is verified in RTL simulation (Verilator) and in real hardware run. We make all the SonicBOOM attacks open-source[1]. Table 2 shows the results of the three attacks, the total number of cycles for reading a single secret byte and the overall throughout of bytes per second for an unprotected core. The cycle count takes into account the entire six phases needed for an attack, described in Section 2.2. The Spectre-v1 and Spectre-v2 have almost the same performance given their similarities in terms of code structure, while the Spectre-v5 code is much simpler and faster–achieving 526 bytes/sec.

[1] https://github.com/sabbaghm/sonicboom-attacks

Table 2: Spectre attacks results on the unprotected machine.

Attack	Cycles for one Byte	Bytes per Second (@100MHz)
Spectre-v1	4857203	20
Spectre-v2	4783403	21
Spectre-v5	196591	526

4.2 Protection Results

Listing 1 and Listing 2 show the printout of Spectre-v1 attack on the unprotected machine and the SSE-rv machine, respectively. We observe similar results for Spectre-v2 and Spectre-v5. The SSE-RV protects the speculative loads from transmitting the secret string out through the data cache.

Listing 1: Spectre-v1 printout on unprotected SonicBOOM.

```
1  want(!) =?= 1.(!) 2.()
2  want(") =?= 1.(") 2.()
3  want(#) =?= 1.(#) 2.()
4  want(S) =?= 1.(S) 2.()
5  want(e) =?= 1.(e) 2.()
6  want(c) =?= 1.(c) 2.()
7  want(r) =?= 1.(r) 2.()
8  want(e) =?= 1.(e) 2.()
9  want(t) =?= 1.(t) 2.()
10 want(I) =?= 1.(I) 2.()
11 want(n) =?= 1.(n) 2.()
12 want(T) =?= 1.(T) 2.()
13 want(h) =?= 1.(h) 2.()
14 want(e) =?= 1.(e) 2.()
15 want(S) =?= 1.(S) 2.()
16 want(o) =?= 1.(o) 2.()
17 want(n) =?= 1.(n) 2.()
18 want(i) =?= 1.(i) 2.()
19 want(c) =?= 1.(c) 2.()
20 want(B) =?= 1.(B) 2.()
21 want(O) =?= 1.(O) 2.()
22 want(O) =?= 1.(O) 2.()
23 want(M) =?= 1.(M) 2.()
```

Listing 2: Spectre-v1 printout on SSE-RV SonicBOOM.

```
want(!) =?= 1.() 2.(|)
want(") =?= 1.() 2.()
want(#) =?= 1.() 2.()
want(S) =?= 1.() 2.()
want(e) =?= 1.() 2.()
want(c) =?= 1.() 2.()
want(r) =?= 1.() 2.(r)
want(e) =?= 1.() 2.()
want(t) =?= 1.() 2.()
want(I) =?= 1.() 2.()
want(n) =?= 1.() 2.()
want(T) =?= 1.() 2.()
want(h) =?= 1.() 2.()
want(e) =?= 1.() 2.()
want(S) =?= 1.() 2.()
want(o) =?= 1.() 2.()
want(n) =?= 1.() 2.()
want(i) =?= 1.() 2.()
want(c) =?= 1.() 2.()
want(B) =?= 1.() 2.()
want(O) =?= 1.() 2.()
want(O) =?= 1.() 2.()
want(M) =?= 1.() 2.()
```

4.3 Performance Impact

The main sources of performance degradation in SSE-RV include: 1) preventing D-cache refills (≈11 cycles) when encountering each transmit Instructions, 2) fencing all memory loads in the Load Queue temporarily, not just the current load, and 3) making all instructions in the PNR windows operate in-order.

We run CoreMark[4] for 10,000 iterations to test the performance of our protected machine. CoreMark is an industry-standard benchmark that measures the performance of single-core CPU. It tests the pipeline operation, cache and memory accesses, and handling of integer operations. The SSE-RV, while protecting against the three Spectre attacks, achieves 0.66 CoreMark/MHz, which is more

than 2.2x faster than a SonicBOOM with out-of-order and speculative execution disabled (0.29 CoreMark/MHz). Compared to the unprotected core with 2.32 CoreMark/MHz, SSE-RV incurs 3.5x slow-down.

We benchmark the SSE-RV core with only conditional branch speculation protection (against Spectre-v1) using the MiBench benchmark suite [10]. We run 500 iterations and report the average execution time. We record the exact number of speculative loads (for conditional branches) for a single iteration using RTL simulation. Fig. 3 shows the benchmarking results for programs with more than 0.01% overhead. Each of these programs has a different instruction mix. The geometric mean of overheads is only 0.53%. The `susan.corners` has the largest overhead of 11.74%, because in this program 40% of all instructions are memory loads, and 0.069% are transmitter instructions (21,252 loads). In contrast, the `bitcount` program has the lowest overhead of 0.02%, because it only has 2.5% of total instructions as memory loads, among which 1.64e-4% (494 loads) are transmitter instructions. We record the exact number of speculative loads using RTL simulation. We find that there is a direct relationship between the ratio of memory loads and transmitter instructions in terms of the performance overhead of SSE-RV. A larger number of transmitter instructions lead to a higher overhead.

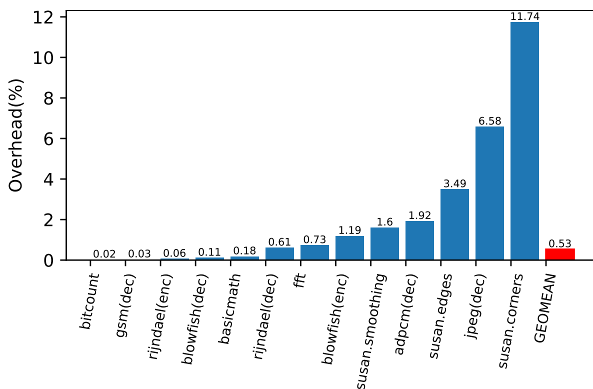

Figure 3: The SSE-RV machine performance overhead.

4.4 Implementation Cost

Table 3 shows the leakage and the dynamic power consumptions for the unprotected and SSE-RV small core, respectively. We found that the total core power consumption increases by only 0.02%.

Table 3: Unprotected vs. SSE-RV core power estimates.

Configuration	Leakage (mW)	Dynamic (mW)	Total (mW)
unprotected	0.2065	286.12	286.33
SSE-RV	0.2071	291.24	291.44

Table 4 shows the NAND2 equivalent gate count and total area for the unprotected and SSE-RV cores. We found that the core gate count increased by only 0.42%.

The resource utilization of the prototype FPGA for the unprotected and SSE-RV cores is also shown in Table 5. In the SSE-RV machine, the Lookup Tables (LUTs) and Flip-flops (FFs) utilizations are increased by only 2.5% and 0.7%, respectively. The increase in LUTs utilization is mainly due to extra logic for taint initialization and propagation phases, while the extra FFs utilization is due to the taint file, the propagation queues, and the intermediate registers.

Table 4: Unprotected vs. SSE-RV core area estimates.

Configuration	Gate count	Total area (μm^2)	Area scaling
Unprotected	509,547	3913320.96	1
SSE-RV	511,693	3929796.48	1.0042

Table 5: Unprotected vs. SSE-RV FPGA resources utilization.

Configuration	LUT	FF	BRAM	DSP
Unprotected	109790	72930	175	36
SSE-RV	112549(+2.5%)	73414(+0.7%)	175	36

5 DISCUSSION

The SSE-RV design presents a general approach to protect against all variants of Spectre attacks when their speculative execution is tracked in the ROB, though it is limited to only cache covert channel. Our modular design of SSE-RV makes adapting it to other attacks easy and dependable. To block other classes of covert channels, the enforcement phase of SSE-RV would require modifications to identify transmitter instructions. For example, to protect against port contention leaks, we could pre-allocate an execution port for the instruction with tainted operands and temporarily serialize accesses to that port until the speculative state is resolved. The TLB covert channel can be blocked by isolating cache ways for transmitter instructions until they are non-speculative.

Due to performing early access permission checks, SonicBOOM is not vulnerable to cross-kernel Meltdown attacks. However, SonicBOOM also contains several other internal buffers, including the line fill buffer, possibly vulnerable against microarchitectural data sampling attacks[24]. Our future work includes extending SSE-RV approach to protect other classes of transient attacks with other covert channels. Further, the performance of SSE-RV can be improved by implementing a local/instruction-targeted fence operation, rather than relying on global fencing, which delays benign loads from accessing the cache temporarily.

6 CONCLUSION

In this work, we propose SSE-RV, an efficient taint tracking based defense mechanism against Spectre attacks. We take a secure-by-design approach and implement our protection scheme in the RISC-V ecosystem. We design all stages of taint tracking in hardware and prototype our SSE-RV processor on an FPGA running Linux. We demonstrate protection against the Spectre-v1, v2, and v5 attacks running on the prototype, while achieving 0.66 CoreMark/MHz. We measure a 0.42% increase in gate count, and a 1.7% increase in total core power consumption of SSE-RV processor relative to an unprotected processor.

7 ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under Grants SaTC1563697, CNS-1916762, and with the industry support from the Center for Hardware and Embedded Systems Security and Trust (CHEST) and Draper Laboratory.

REFERENCES

[1] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* (2020), 10–21.

[2] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *Proc. Conf. Computer and Communications Security*. 785–800.

[3] Claudio Canella, Khaled N. Khasawneh, and Daniel Gruss. 2020. The Evolution of Transient-Execution Attacks. In *Proc. Great Lakes Symp. VLSI*. 163–168.

[4] Embedded Microprocessor Benchmark Consortium. 2021. CoreMark. https://github.com/eembc/coremark.

[5] Advanced Micro Devices. 2018. *Software techniques for managing speculation on AMD processors*. Technical Report.

[6] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*. 693–707.

[7] Cheng Gongye, Yunsi Fei, and Thomas Wahl. 2020. Reverse-Engineering Deep Neural Networks Using Floating-Point Timing Side-Channels. In *Design Automation Conference*. 1–6.

[8] Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis, and Krste Asanovic. 2019. Replicating and Mitigating Spectre Attacks on a Open Source RISC-V Microarchitecture. (2019).

[9] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*. 955–972.

[10] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. Int. Workshop Workload Characterization. WWC-4 (Cat. No.01EX538)*. 3–14.

[11] John Hennessy and David Patterson. 2017. *Computer Architecture, Sixth Edition, A Quantitative Approach*. Morgan Kaufmann.

[12] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *Int. Symp. Microarchitecture*. 974–987.

[13] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01203

[14] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*. 557–574.

[15] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[16] Jason Lowe-Power et al. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152

[17] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Karg, and Daniel Gruss. 2020. ConTExT: A Generic Approach for Mitigating Spectre. In *Network and Distributed Systems Security Symp.*

[18] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *Computer Security – ESORICS*. 279–299.

[19] SUSE Linux Enterprise Server. 2018. *Security update for kernel-firmware*. Technical Report. https://www.suse.com/support/update/announcement/2018/suse-su-20180008-1

[20] SUSE Linux Enterprise Server. 2020. *Security Vulnerability: Spectre side channel attack "Lazy FPU Save/Restore" aka CVE-2018-3665*. Technical Report 7023076. https://www.suse.com/support/kb/doc/?id=000019231

[21] André Seznec and Pierre Michaud. 2006. A case for (partially) TAgged GEometric history length branch prediction. *J. Instr. Level Parallelism* 8 (2006).

[22] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*. 395–410.

[23] Eugene Tarassov. 2021. Xilinx Vivado block designs for FPGA RISC-V SoC running Debian Linux distro. https://github.com/eugene-tarassov/vivado-risc-v.

[24] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.

[25] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proc. Int. Symp. Microarchitecture*. 428–441.

[26] Y. Yarom and K. Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Sec. Symp.* 719–732.

[27] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W Fletcher. 2020. Speculative Data-Oblivious Execution: Mobilizing Safe Prediction For Safe and Efficient Speculative Execution. In *Proc. Int. Symp. Computer Architecture*. 707—720.

[28] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proc. Int. Symp. Microarchitecture*. 954–968.

[29] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. (2020).