SSE-RV: Secure Speculative Execution via RISC-V Open Hardware Design

<u>Majid Sabbagh</u>, Yunsi Fei, David Kaeli

Department of Electrical and Computer Engineering

Northeastern University, Boston, MA, USA

Fifth Workshop on Computer Architecture Research with RISC-V (CARRV 2021)

Northeastern University

A sample uarch: Intel Skylake

Closed/propriety hardware design

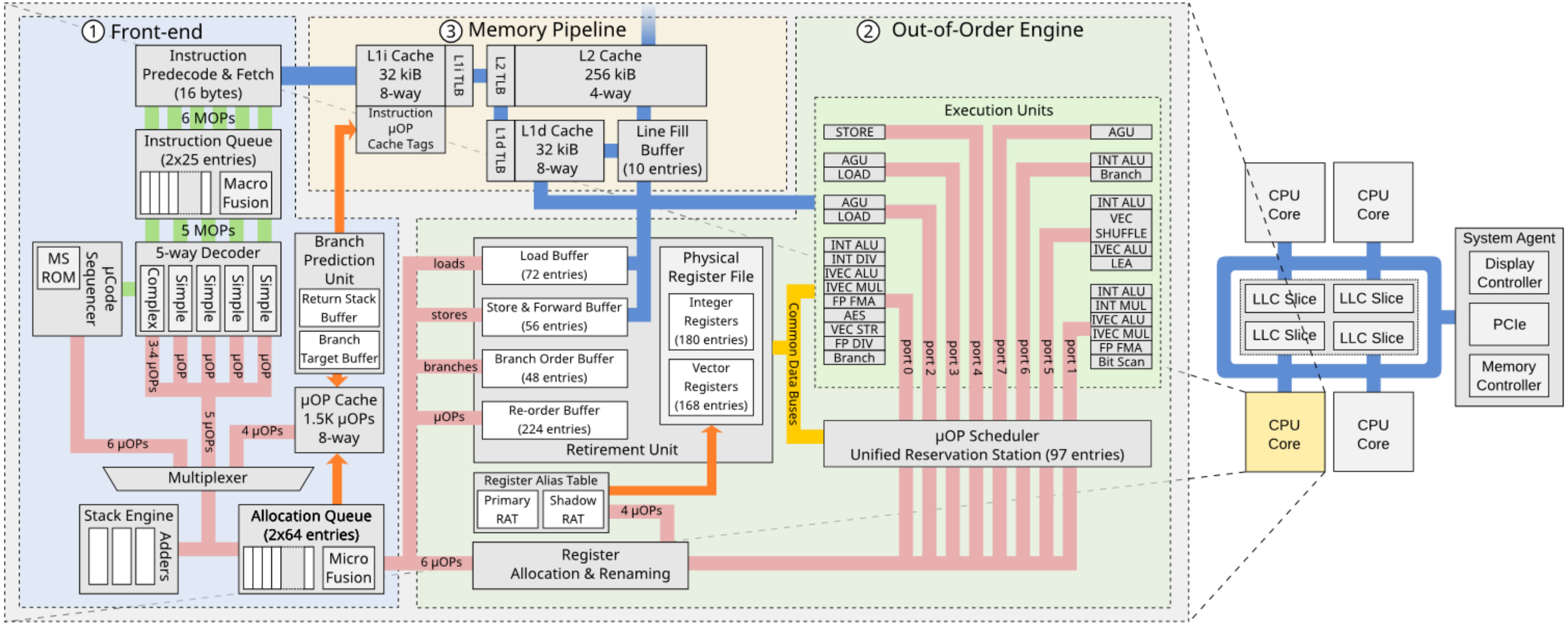
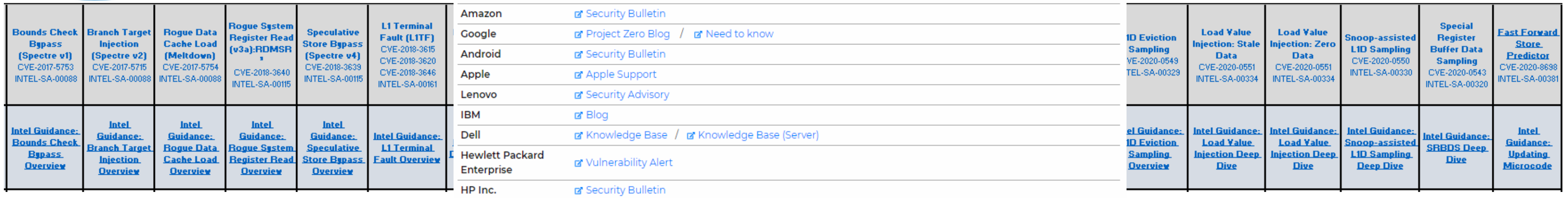
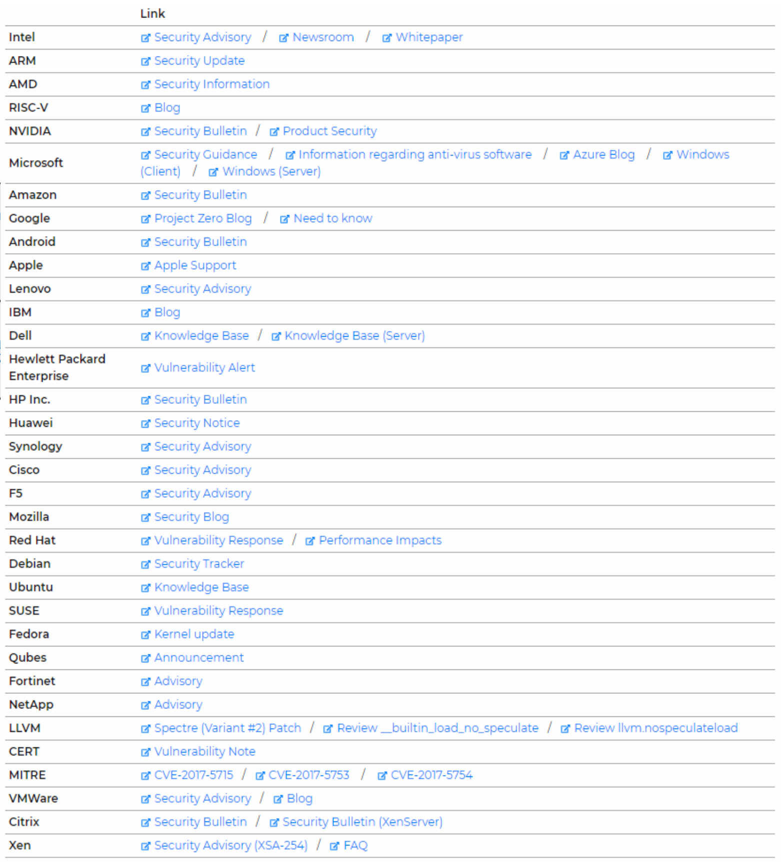


Figure by Stephan van Schaik, *the VU*

Transient-Execution attacks: why do they matter?



Bounds Check Bypass (Spectre v1) CVE-2017-5753 INTEL-SA-00088	Branch Target Injection (Spectre v2) CVE-2017-5715 INTEL-SA-00088	Rogue Data Cache Load (Meltdown) CVE-2017-5754 INTEL-SA-00088	Rogue System Register Read (v3a):RDMSR, CVE-2018-3640 INTEL-SA-00115	Speculative Store Bypass (Spectre v4) CVE-2018-3639 INTEL-SA-00115	L1 Terminal Fault (L1TF) CVE-2018-3615 CVE-2018-3620 CVE-2018-3646 INTEL-SA-00161	1D Eviction Sampling VE-2020-0549 TEL-SA-00329	Load Value Injection: Stale Data CVE-2020-0551 INTEL-SA-00334	Load Value Injection: Zero Data CVE-2020-0551 INTEL-SA-00334	Snoop-assisted L1D Sampling CVE-2020-0550 INTEL-SA-00330	Special Register Buffer Data Sampling CVE-2020-0543 INTEL-SA-00320	Fast Forward Store Predictor CVE-2020-8698 INTEL-SA-00381
Intel Guidance: Bounds Check Bypass Overview	Intel Guidance: Branch Target Injection Overview	Intel Guidance: Rogue Data Cache Load Overview	Intel Guidance: Rogue System Register Read Overview	Intel Guidance: Speculative Store Bypass Overview	Intel Guidance: L1 Terminal Fault Overview	el Guidance: 1D Eviction Sampling Overview	Intel Guidance: Load Value Injection Deep Dive	Intel Guidance: Load Value Injection Deep Dive	Intel Guidance: Snoop-assisted L1D Sampling Deep Dive	Intel Guidance: SRBDS Deep Dive	Intel Guidance: Updating Microcode



	Link
Intel	Security Advisory / Newsroom / Whitepaper
ARM	Security Update
AMD	Security Information
RISC-V	Blog
NVIDIA	Security Bulletin / Product Security
Microsoft	Security Guidance / Information regarding anti-virus software / Azure Blog / Windows (Client) / Windows (Server)
Amazon	Security Bulletin
Google	Project Zero Blog / Need to know
Android	Security Bulletin
Apple	Apple Support
Lenovo	Security Advisory
IBM	Blog
Dell	Knowledge Base / Knowledge Base (Server)
Hewlett Packard Enterprise	Vulnerability Alert
HP Inc.	Security Bulletin
Huawei	Security Notice
Synology	Security Advisory
Cisco	Security Advisory
F5	Security Advisory
Mozilla	Security Blog
Red Hat	Vulnerability Response / Performance Impacts
Debian	Security Tracker
Ubuntu	Knowledge Base
SUSE	Vulnerability Response
Fedora	Kernel update
Qubes	Announcement
Fortinet	Advisory
NetApp	Advisory
LLVM	Spectre (Variant #2) Patch / Review __builtin_load_no_speculate / Review llvm.nospeculateload
CERT	Vulnerability Note
MITRE	CVE-2017-5715 / CVE-2017-5753 / CVE-2017-5754
VMWare	Security Advisory / Blog
Citrix	Security Bulletin / Security Bulletin (XenServer)
Xen	Security Advisory (XSA-254) / FAQ

meltdownattack.com

Highlights

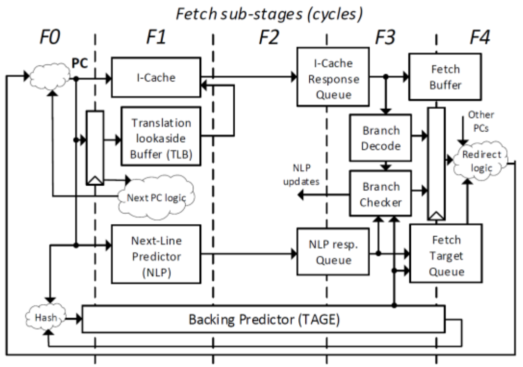
- **Implemented a novel taint tracking architecture**
  - Based on *SonicBOOM*
  - **SSE-RV protects against Spectre attacks**, outperforming the state-of-the-art
- Developed an **FPGA prototype for the proposed SSE-RV**
- **Evaluated the security guarantees** delivered by our protection scheme, as well as **area/power/performance overheads**

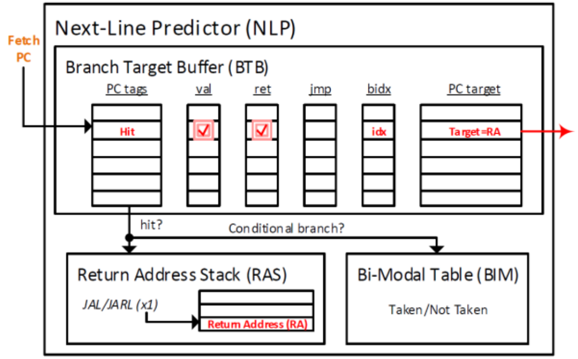
Outline

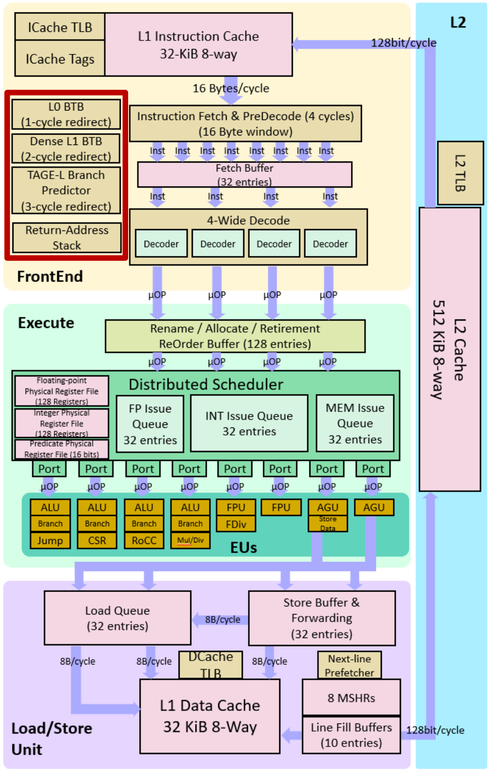
- ❑SonicBOOM microarchitecture and its speculative execution vulnerabilities
- ❑Spectre attacks and state of the art defenses
- ❑SSE-RV architecture and mechanism
- ❑Experimental results and analysis
- ❑Future work

SonicBOOM microarchitecture

- 3rd-gen Berkeley Out-of-order Machine (SonicBOOM)[1]
  - Out-of-order and speculative features allow for transient execution
  - Comparable features to commercial processors



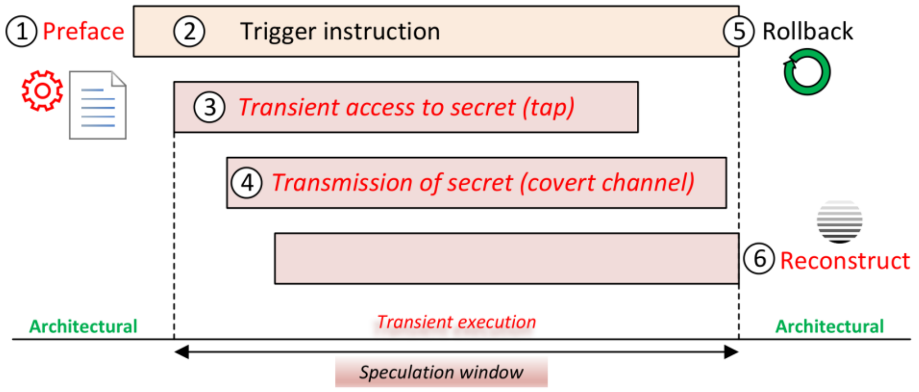



ICache TLB
ICache Tags
L1 Instruction Cache 32-KiB 8-way
128bit/cycle
L2
16 Bytes/cycle
L0 BTB (1-cycle redirect)
Dense L1 BTB (2-cycle redirect)
TAGE-L Branch Predictor (3-cycle redirect)
Return-Address Stack
Instruction Fetch & PreDecode (4 cycles) (16 Byte window)
Inst
Fetch Buffer (32 entries)
4-Wide Decode
Decoder
FrontEnd
L2 TLB
L2 Cache 512 KiB 8-way
µOP
Execute
Rename / Allocate / Retirement ReOrder Buffer (128 entries)
Floating-point Physical Register File (128 Registers)
Integer Physical Register File (128 Registers)
Predicate Physical Register File (16 bits)
Distributed Scheduler
FP Issue Queue 32 entries
INT Issue Queue 32 entries
MEM Issue Queue 32 entries
Port
ALU
Branch
Jump
CSR
RoCC
Mul/Div
FPU
FDiv
AGU
Store Data
EUs
Load Queue (32 entries)
8B/cycle
Store Buffer & Forwarding (32 entries)
DCache TLB
Next-line Prefetcher
L1 Data Cache 32 KiB 8-Way
8 MSHRs
Line Fill Buffers (10 entries)
Load/Store Unit

[1] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, Krste., CARRV'20

Spectre attacks

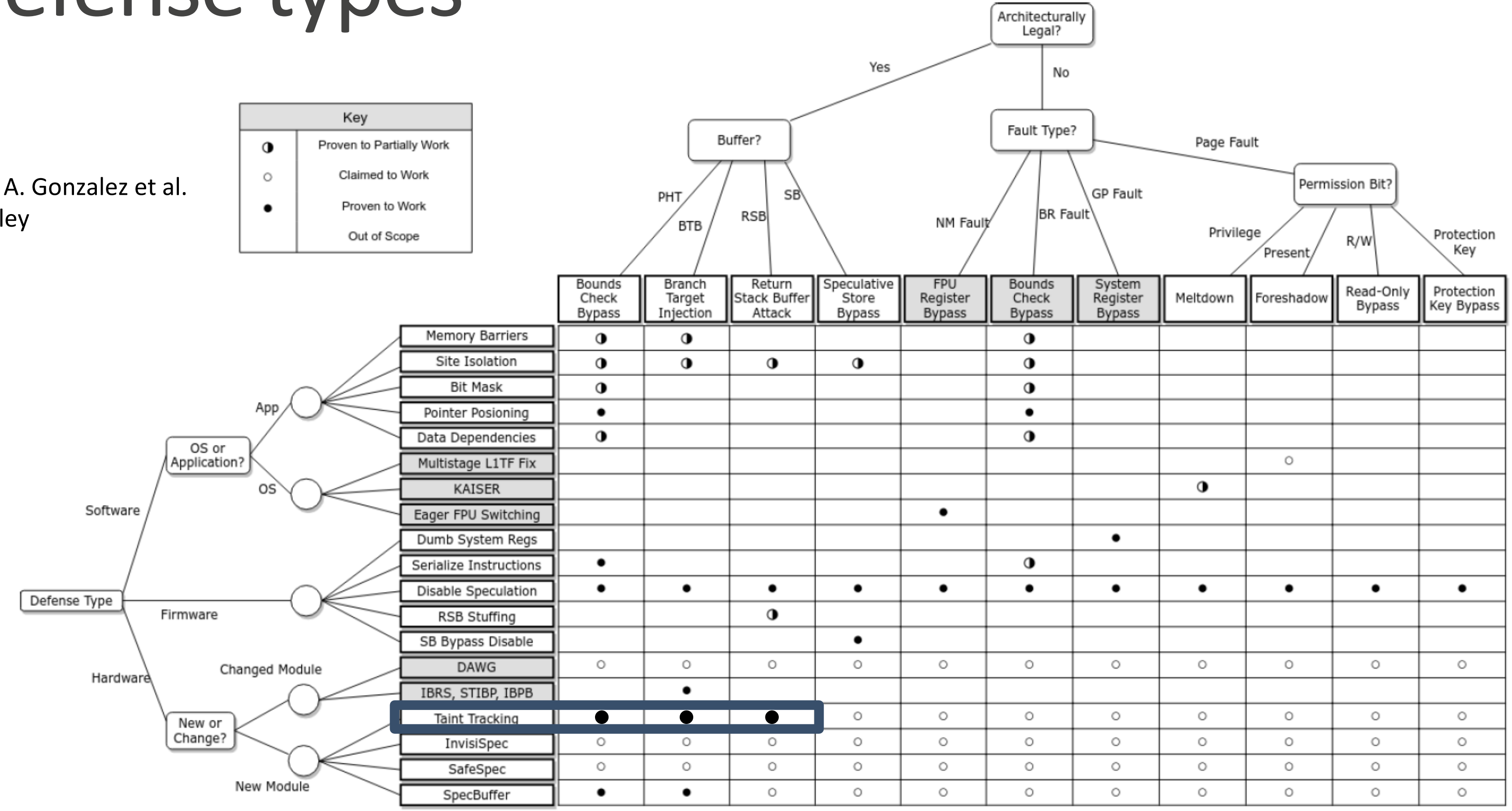
Transient execution attacks, including Spectre attacks, can be described in six major phases:



	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

Defense types

Figure by A. Gonzalez et al.
UC Berkeley



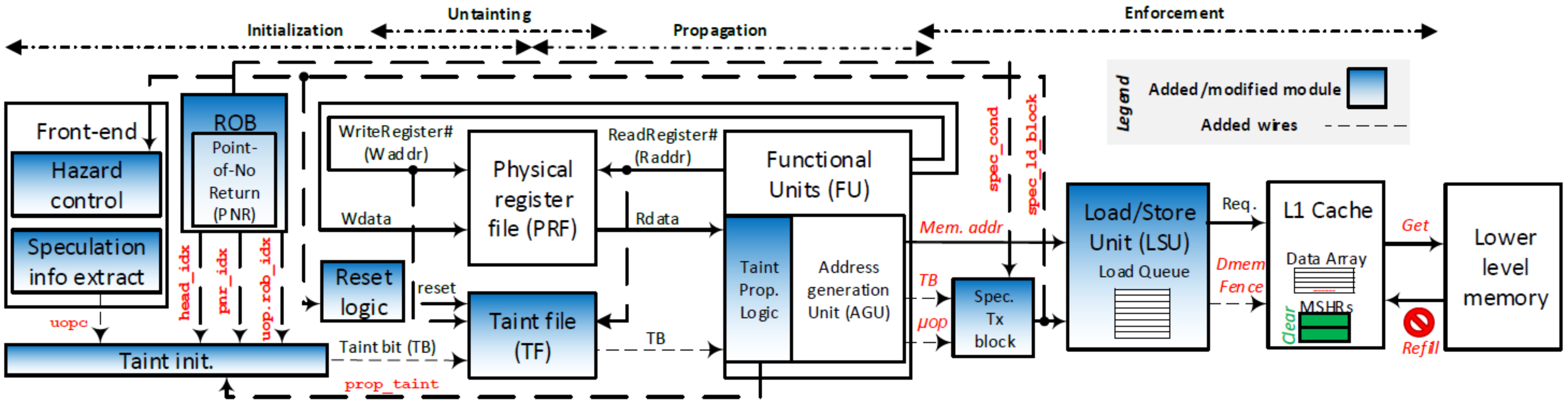
Key	
◑	Proven to Partially Work
○	Claimed to Work
●	Proven to Work
	Out of Scope

	Bounds Check Bypass	Branch Target Injection	Return Stack Buffer Attack	Speculative Store Bypass	FPU Register Bypass	Bounds Check Bypass	System Register Bypass	Meltdown	Foreshadow	Read-Only Bypass	Protection Key Bypass
Memory Barriers	◑	◑				◑					
Site Isolation	◑	◑	◑	◑		◑					
Bit Mask	◑					◑					
Pointer Posioning	●					●					
Data Dependencies	◑					◑					
Multistage L1TF Fix									○		
KAISER								◑			
Eager FPU Switching					●						
Dumb System Regs							●				
Serialize Instructions	●					◑					
Disable Speculation	●	●	●	●	●	●	●	●	●	●	●
RSB Stuffing			◑								
SB Bypass Disable				●							
DAWG	○	○	○	○	○	○	○	○	○	○	○
IBRS, STIBP, IBPB		●									
Taint Tracking	●	●	●	○	○	○	○	○	○	○	○
InvisiSpec	○	○	○	○	○	○	○	○	○	○	○
SafeSpec	○	○	○	○	○	○	○	○	○	○	○
SpecBuffer	●	●	○	○	○	○	○	○	○	○	○

State of the art defenses

- Speculative Taint Tracking (STT)
  - A **hardware-level taint tracking** scheme
  - Taints data that is accessed during a speculative window and **delays** any subsequent **load** with tainted operands until they become untainted
  - **Untainting** of registers happens **once the instruction that invoked it becomes non-speculative**
- Context-Sensitive Fencing (CSF)
  - A **micro-code level defense** against Spectre attacks
  - **Dynamically alters the decoding** of the instruction stream, **injecting fences** only when dynamic conditions indicate they are needed

SSE-RV architectural overview



Point of No Return (PNR)

- Speculation condition is determined by *fast* PNR[2]:

ReOrder Buffer (ROB)

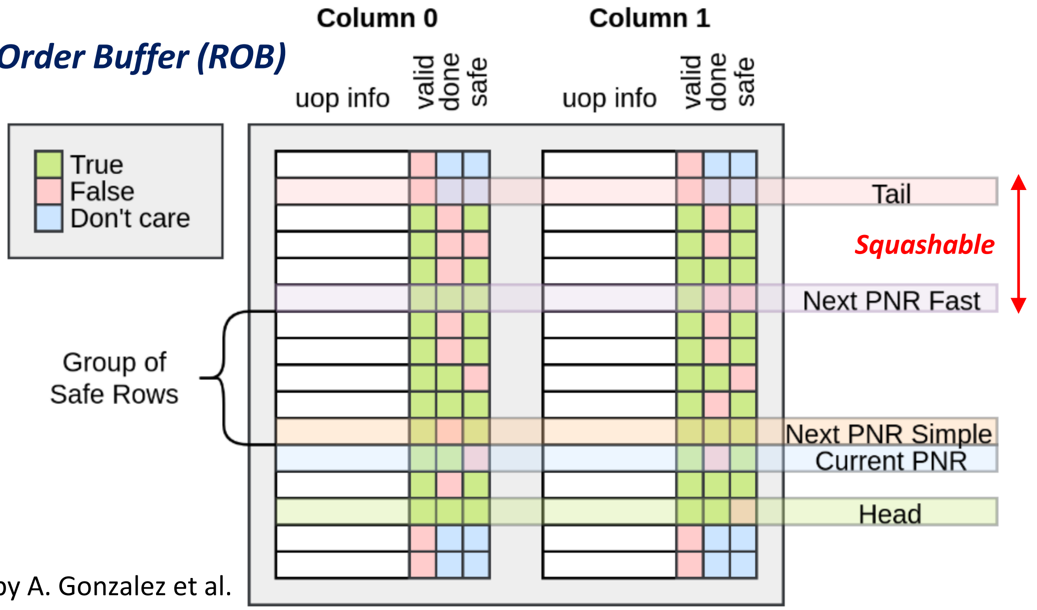
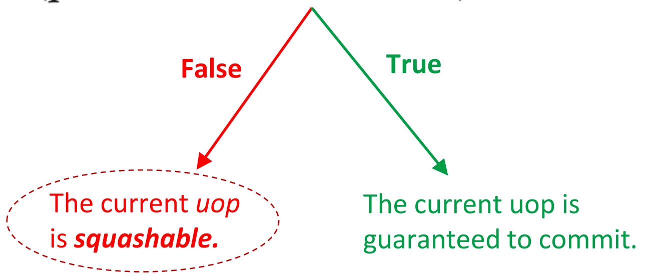


Figure by A. Gonzalez et al. UC Berkeley

PNR equation

$$(uop.rob\_idx < pnr\_idx) \oplus$$
$$(uop.rob\_idx < head\_idx) \oplus$$
$$(pnr\_idx < head\_idx)$$


False
True
The current *uop* is ***squashable.***
The current uop is guaranteed to commit.

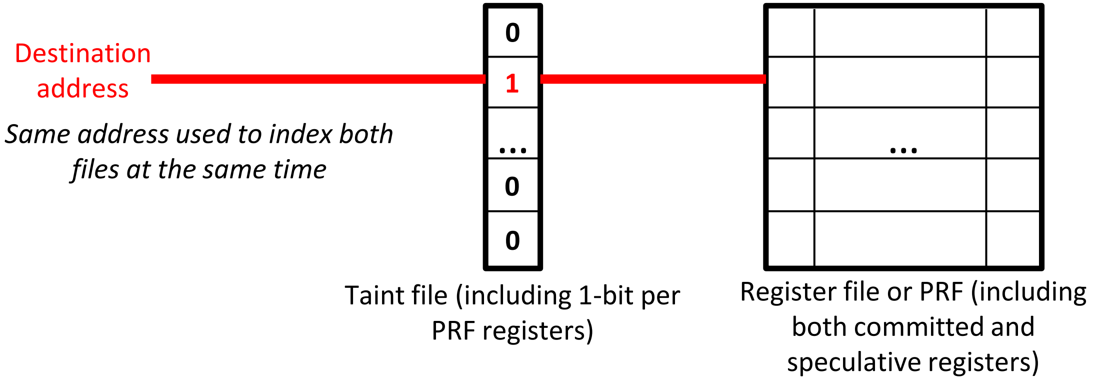
All forms of speculation is covered by PNR!

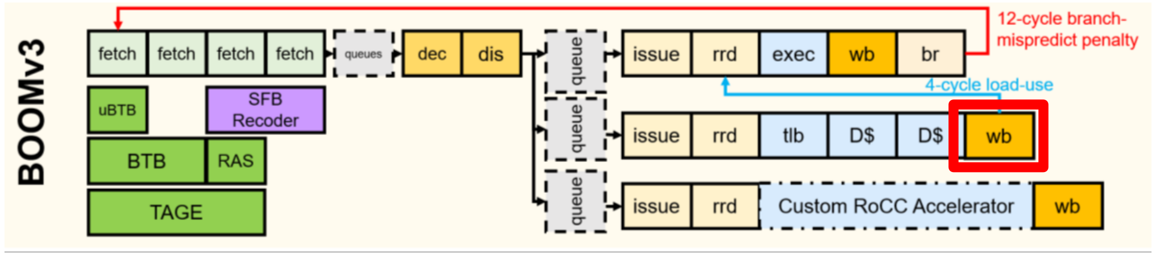
[2] A. Gonzalez, B. Korpan, J. Zhao, E. Younis, and K. Asanovic, Krste., CARRV'19

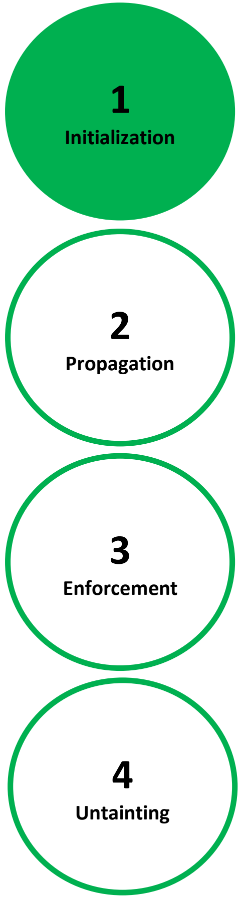
SSE-RV implementation - Initialization

Squashable (≈speculative) **access** (load) instruction's **destination register**

- **Squashable loads:** *the load is younger than PNR* → **taint destination register**
- **Set the corresponding bit in taint file at the write-back stage** for every squashable load

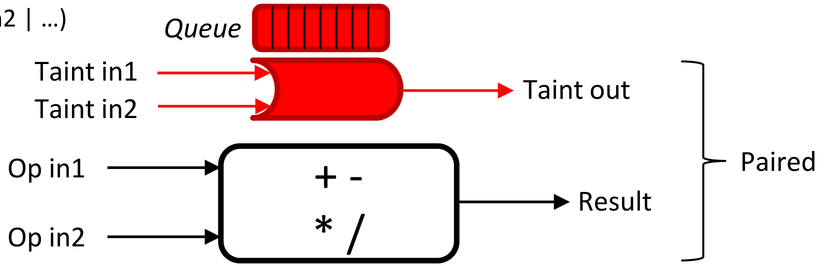

Destination address
Same address used to index both files at the same time
0
1
...
0
0
...
Taint file (including 1-bit per PRF registers)
Register file or PRF (including both committed and speculative registers)

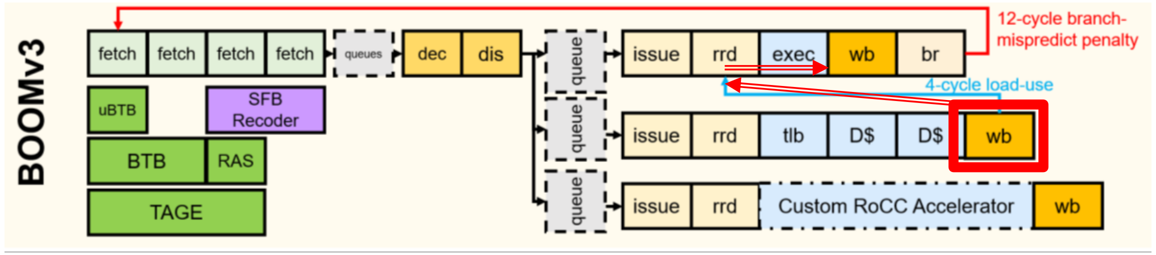


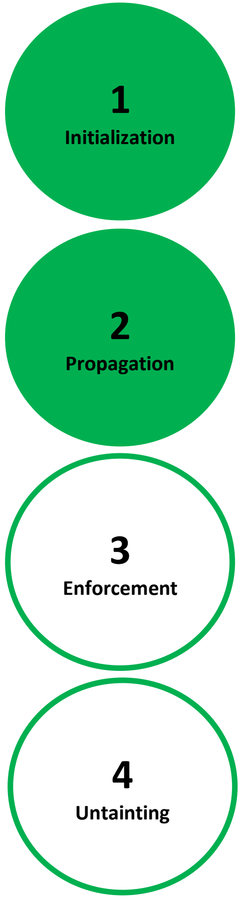


SSE-RV implementation - Propagation

- **Taints**:
  - Output of speculative access instructions
  - Output of instructions with tainted inputs
- **Propagation in parallel to functional units** through 1-bit extended pipeline registers (chain of instructions) and synchronization queues
  - Every in-flight instruction have #operands 1-bit taint inputs and a 1-bit taint output
    - Taint_out = (taint_in1 | taint_in2 | …)

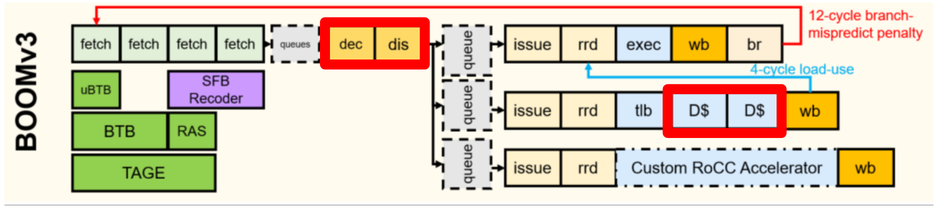


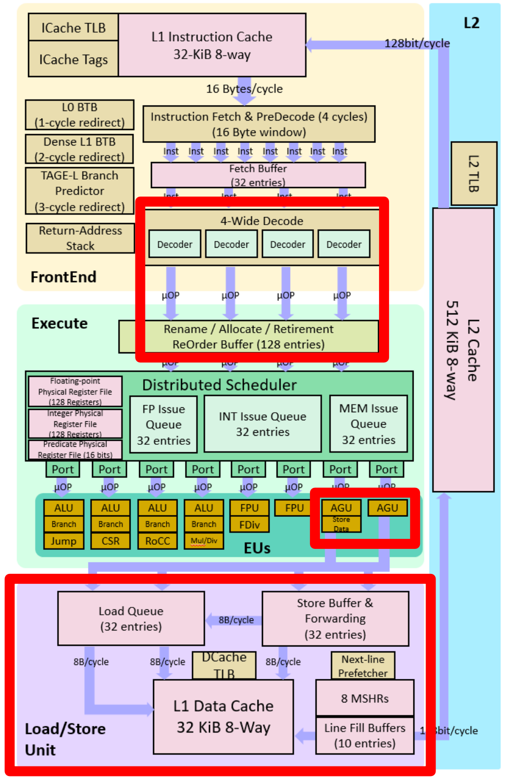


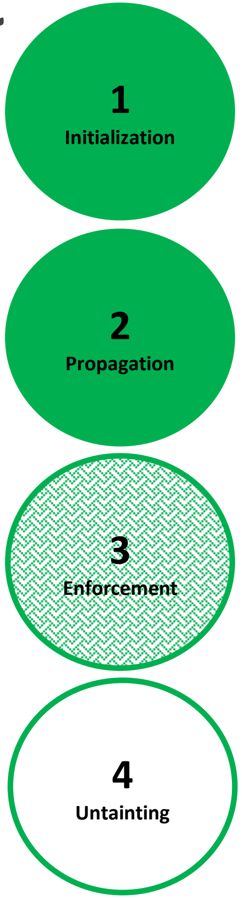


SSE-RV implementation - Enforcement

- **Detect transmit instructions (loads with tainted addresses)** at the ***memaddrcalc (AGU)*** stage
- **Delay load to cache** until all operands are untainted (e.g., load address)
  - **Block speculative cache refills**
  - **Backpressure on the dispatch units**
    - Introduce a *dis* hazard to stall the pipeline, until the branch is resolved
  - Ops untainted → proceed with cache refills

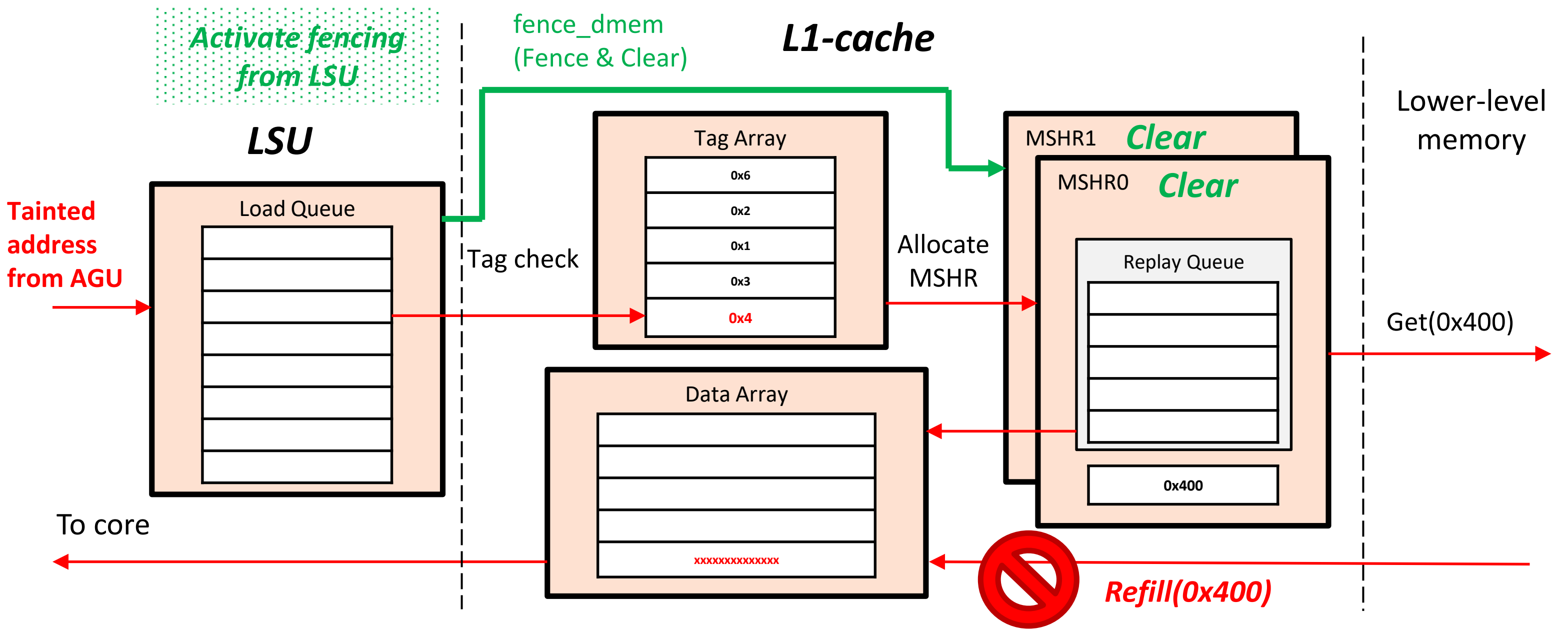


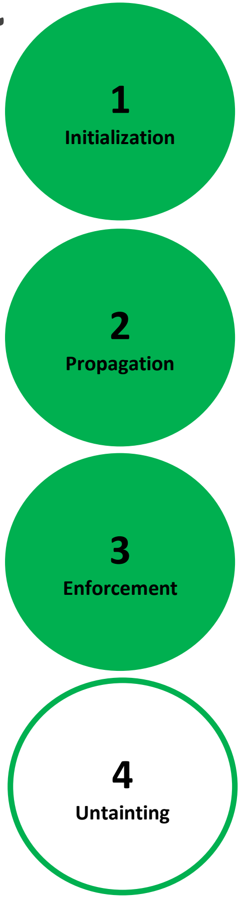

ICache TLB
ICache Tags
L1 Instruction Cache 32-KiB 8-way
128bit/cycle
L2
16 Bytes/cycle
L0 BTB (1-cycle redirect)
Dense L1 BTB (2-cycle redirect)
TAGE-L Branch Predictor (3-cycle redirect)
Return-Address Stack
Instruction Fetch & PreDecode (4 cycles) (16 Byte window)
Fetch Buffer (32 entries)
Inst
4-Wide Decode
Decoder
FrontEnd
µOP
L2 TLB
L2 Cache 512 KiB 8-way
Execute
Rename / Allocate / Retirement ReOrder Buffer (128 entries)
Floating-point Physical Register File (128 Registers)
Integer Physical Register File (128 Registers)
Predicate Physical Register File (16 bits)
Distributed Scheduler
FP Issue Queue 32 entries
INT Issue Queue 32 entries
MEM Issue Queue 32 entries
Port
ALU
Branch
Jump
CSR
RoCC
Mul/Div
FPU
FDiv
AGU
Store Data
EUs
Load Queue (32 entries)
Store Buffer & Forwarding (32 entries)
8B/cycle
DCache TLB
Next-line Prefetcher
8 MSHRs
L1 Data Cache 32 KiB 8-Way
Line Fill Buffers (10 entries)
Load/Store Unit
1 bit/cycle



SSE-RV implementation - Enforcement

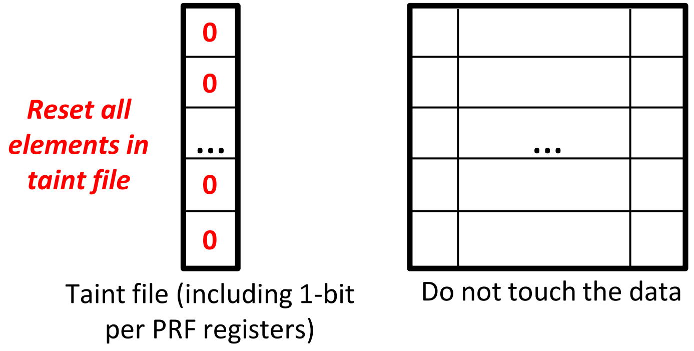
- **Enable fencing to:**
  - **Clear all** the MSHRs and prefetch buffers
  - **Block the refill** until the transmit instruction is no longer speculative

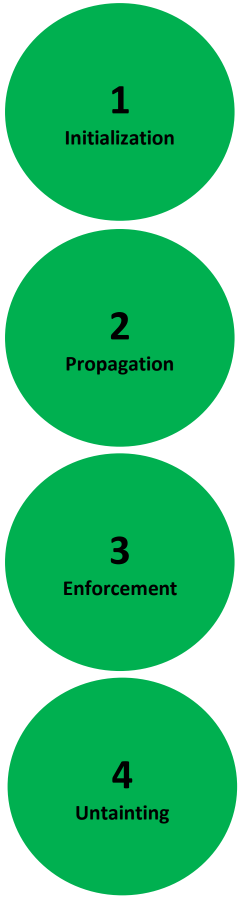




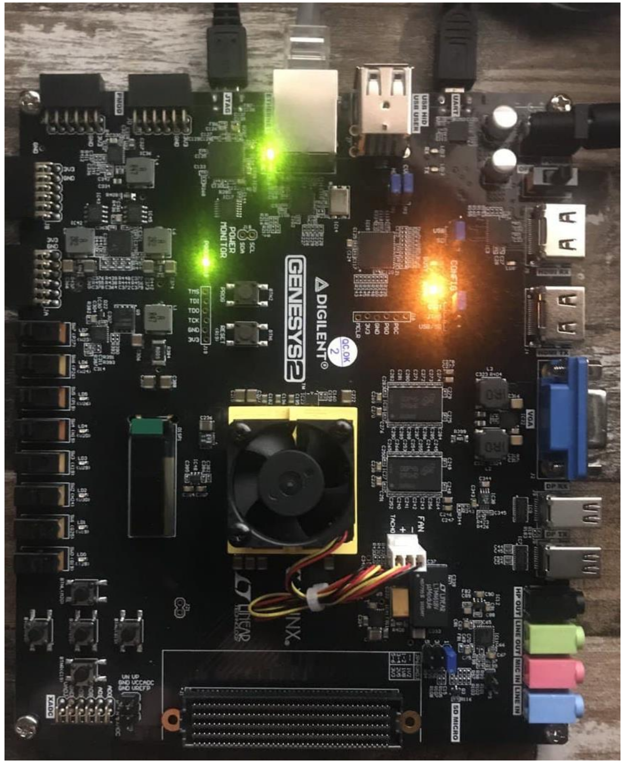
SSE-RV implementation - Untainting

- **Disable the fencing and start untainting:**
  - A speculative access instruction becomes non-speculative (branch is resolved)
  - An instruction has all of its inputs untainted
- In case of misprediction, **the data and the corresponding taint 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**
  - *Assuming there is only one batch of in-flight speculative instructions*

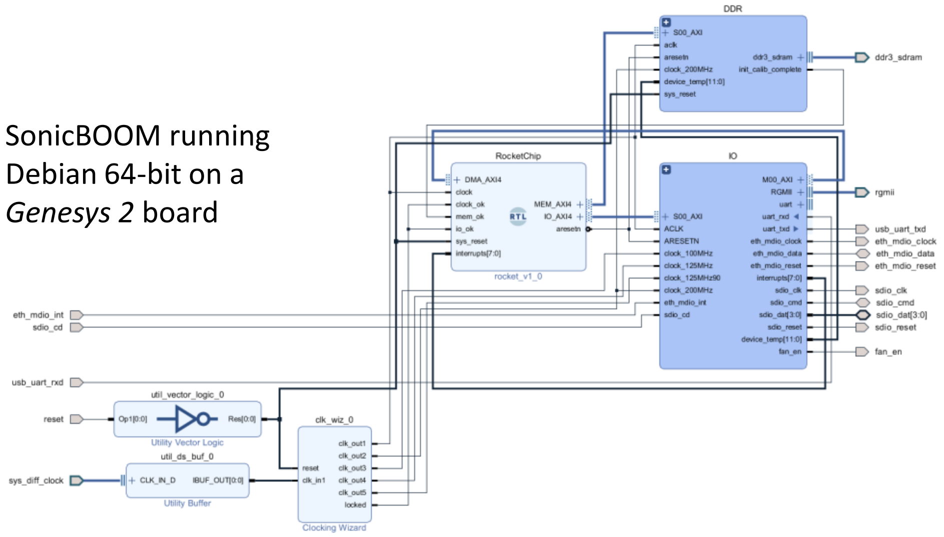

Reset all elements in taint file
0
0
...
0
0
...
Taint file (including 1-bit per PRF registers)
Do not touch the data



Prototyping



SonicBOOM running Debian 64-bit on a *Genesys 2* board



- Kintex-7 XC7K325T-2FFG900C on Genesys-2 board
- Core frequency = *100MHz*
- Small core machine
  - 1-wide, 3-issue, 16KB D$ (2 MSHRs), 16KB I$, 512KB L2$

SonicBOOM Spectre Attacks

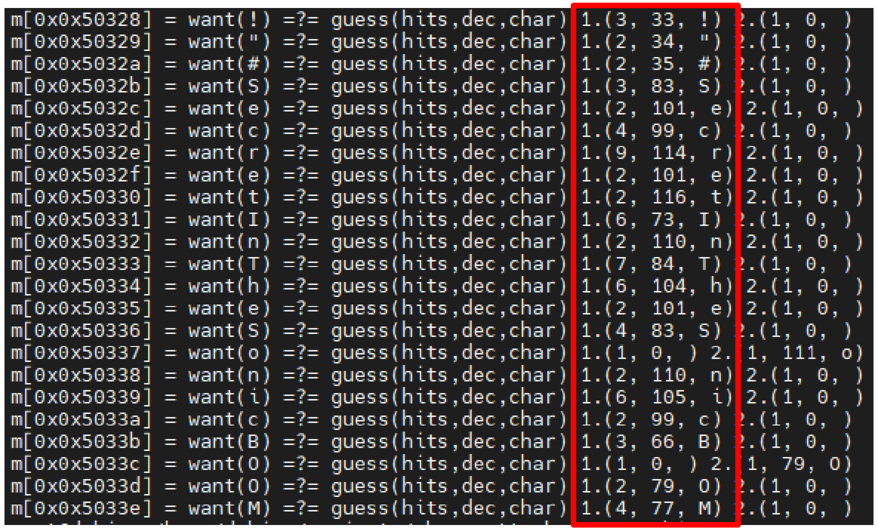
- Replicated Spectre-v1, v2, and v5 on SonicBOOM and open sourced at:

`https://chest.coe.neu.edu/?current_page=SOURCE_CODE`

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

Protection results

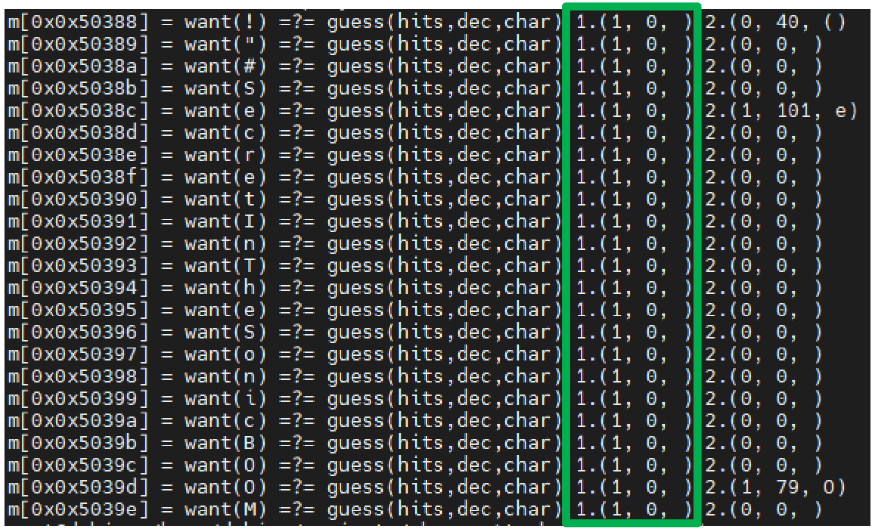
Spectre-v1 on Original SonicBOOM



```
m[0x0x50328] = want(!) =?= guess(hits,dec,char) 1.(3, 33, !) 2.(1, 0, )
m[0x0x50329] = want(") =?= guess(hits,dec,char) 1.(2, 34, ") 2.(1, 0, )
m[0x0x5032a] = want(#) =?= guess(hits,dec,char) 1.(2, 35, #) 2.(1, 0, )
m[0x0x5032b] = want(S) =?= guess(hits,dec,char) 1.(3, 83, S) 2.(1, 0, )
m[0x0x5032c] = want(e) =?= guess(hits,dec,char) 1.(2, 101, e) 2.(1, 0, )
m[0x0x5032d] = want(c) =?= guess(hits,dec,char) 1.(4, 99, c) 2.(1, 0, )
m[0x0x5032e] = want(r) =?= guess(hits,dec,char) 1.(9, 114, r) 2.(1, 0, )
m[0x0x5032f] = want(e) =?= guess(hits,dec,char) 1.(2, 101, e) 2.(1, 0, )
m[0x0x50330] = want(t) =?= guess(hits,dec,char) 1.(2, 116, t) 2.(1, 0, )
m[0x0x50331] = want(I) =?= guess(hits,dec,char) 1.(6, 73, I) 2.(1, 0, )
m[0x0x50332] = want(n) =?= guess(hits,dec,char) 1.(2, 110, n) 2.(1, 0, )
m[0x0x50333] = want(T) =?= guess(hits,dec,char) 1.(7, 84, T) 2.(1, 0, )
m[0x0x50334] = want(h) =?= guess(hits,dec,char) 1.(6, 104, h) 2.(1, 0, )
m[0x0x50335] = want(e) =?= guess(hits,dec,char) 1.(2, 101, e) 2.(1, 0, )
m[0x0x50336] = want(S) =?= guess(hits,dec,char) 1.(4, 83, S) 2.(1, 0, )
m[0x0x50337] = want(o) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(1, 111, o)
m[0x0x50338] = want(n) =?= guess(hits,dec,char) 1.(2, 110, n) 2.(1, 0, )
m[0x0x50339] = want(i) =?= guess(hits,dec,char) 1.(6, 105, i) 2.(1, 0, )
m[0x0x5033a] = want(c) =?= guess(hits,dec,char) 1.(2, 99, c) 2.(1, 0, )
m[0x0x5033b] = want(B) =?= guess(hits,dec,char) 1.(3, 66, B) 2.(1, 0, )
m[0x0x5033c] = want(O) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(1, 79, O)
m[0x0x5033d] = want(O) =?= guess(hits,dec,char) 1.(2, 79, O) 2.(1, 0, )
m[0x0x5033e] = want(M) =?= guess(hits,dec,char) 1.(4, 77, M) 2.(1, 0, )
```

Secret extracted!

Spectre-v1 on SonicBOOM with SSE-RV



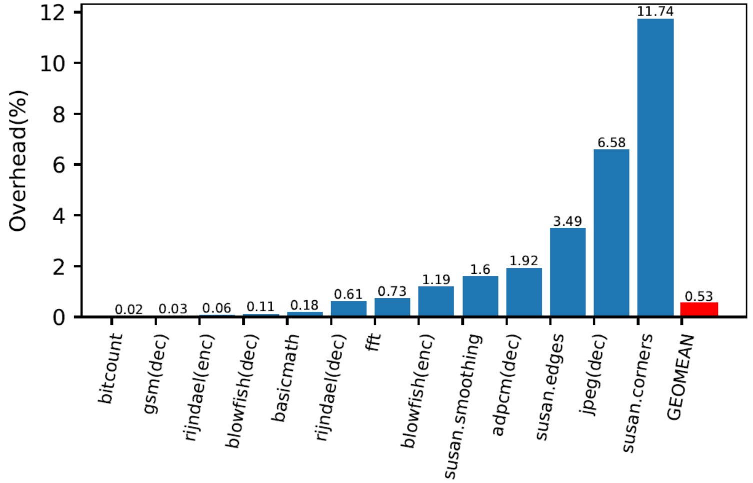
```
m[0x0x50388] = want(!) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 40, ()
m[0x0x50389] = want(") =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x5038a] = want(#) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x5038b] = want(S) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x5038c] = want(e) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(1, 101, e)
m[0x0x5038d] = want(c) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x5038e] = want(r) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x5038f] = want(e) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x50390] = want(t) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x50391] = want(I) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x50392] = want(n) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x50393] = want(T) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x50394] = want(h) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x50395] = want(e) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x50396] = want(S) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x50397] = want(o) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x50398] = want(n) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x50399] = want(i) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x5039a] = want(c) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x5039b] = want(B) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x5039c] = want(O) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
m[0x0x5039d] = want(O) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(1, 79, O)
m[0x0x5039e] = want(M) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(0, 0, )
```

Secret hidden!

Similar results for Spectre-v2 and v5

Performance impact

Configuration	CoreMark Size	Total cycles	Total time (secs)	Iterations/Sec (CoreMark)	CoreMark/MHz	Iterations
Original small core SonicBoom machine	666	43,127	43.1	**231.9**	**2.32**	**10,000**
SSE-RV protected small core SonicBoom machine	666	153,992	154.0	**64.9**	**0.65**	**10,000**



The MiBench[3] performance overhead *when only protecting against Spectre-v1*

More transmitter instructions → more overhead!

[3] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown, MiBench: A free, commercially representative embedded benchmark suite, WWC-4

Implementation Cost

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

Unprotected vs. SSE-RV core area estimates, 130nm BiCMOS technology

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

Unprotected vs. SSE-RV core power estimates, 130nm BiCMOS technology

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

Future work

- ❑ **Enhance the SSE-RV performance** in both single and multi-core settings
- ❑ **Explore the vulnerability** of SonicBOOM to **MDS attacks**
- ❑ **Extend** our methodology to defend **against software attacks**

Thank you!

Comments/Questions?

sabbagh.m@northeastern.edu

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.