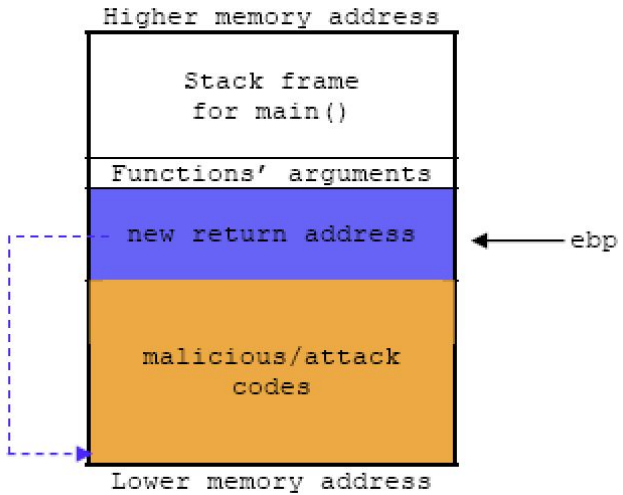
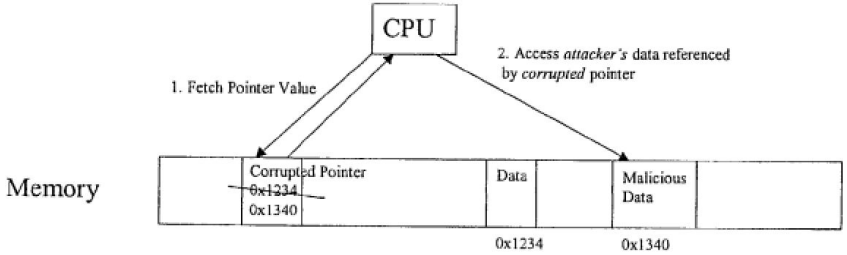
ThundaTag: Disparate Domain Tagging to Enforce Benign Program Behavior

By Shibo Chen and Alex Kisil

Motivation

Many attacks rely on the ability of forging pointers
Solutions: Rule-based Systems, Finer-grained encryption, Memory Space Randomization etc.

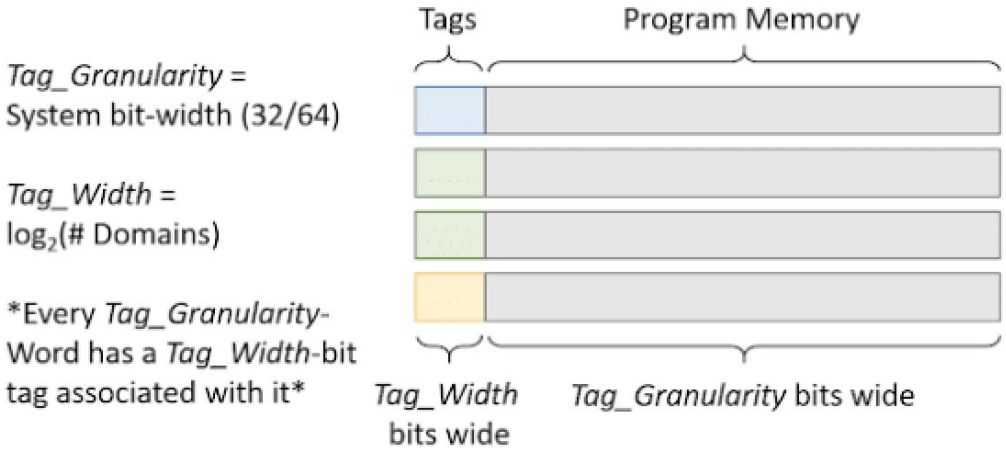




https://www.tenouk.com/Bufferoverflowc/Bufferoverflow4.html
https://patentimages.storage.googleapis.com/e7/9b/6e/245e7ec8fff374/US7752459.pdf

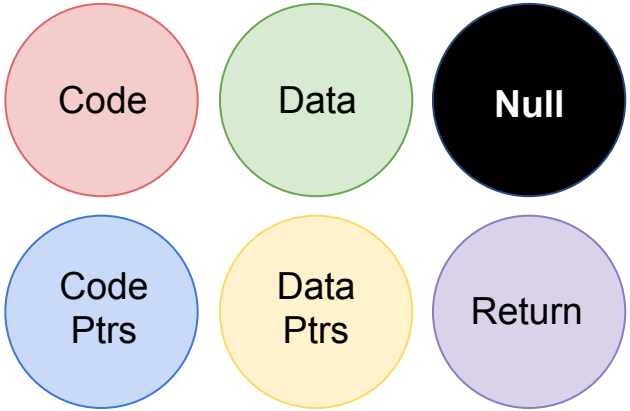
Methodology

- Architectural support to impose strict checks and propagation rules on instructions, i.e. the program should only jump on code pointers, and not data pointers or data



Methodology Ctnd.

- Disparate Domains
  - 6 domains: code, data, code pointer, data pointer, return, null (4 bits for easy alignment)



Rules for Control Flow and Arithmetic Operations

- Only code can be executed.
- Can only jump/branch on code pointers.
- Can only return on return type.
- Return can only be returned. Any other operation is an exception.
- NULL can not be accessed under any circumstances.

Propagations Rules in ALU and Memory

B \ A	-	-	-	-
-	O	O	O	O
-	O	O	O	O
-	O	O	O	O
-	O	O	O	O

Store (overriden)
DEST: MEM[RS1 + offset]

B \ A	-	-	-	-
-	O	O	O	O
-	O	O	O	O
-	O	O	O	O
-	O	O	O	O

Non-Store ops (overriden)
DEST: RD

B \ A	D	DP	C	CP
D	!!!	D	!!!	!!!
DP	!!!	DP	!!!	!!!
C	!!!	!!!	!!!	!!!
CP	!!!	CP	!!!	!!!

Load (B = MEM[RS1+offset])
DEST: RD

B \ A	D	DP	C	CP
D	!!!	D	!!!	!!!
DP	!!!	DP	!!!	!!!
C	!!!	!!!	!!!	!!!
CP	!!!	CP	!!!	!!!

Store
DEST: MEM[RS1 + offset]

B \ A	D	DP	C	CP
D	D	DP	!!!	!
DP	DP	!	!!!	!
C	!!!	!!!	!!!	!!!
CP	!	!	!!!	!

Reg-Reg Arith (ADD only)
DEST: RD

B \ A	D	DP	C	CP
D	D	DP	!!!	!
DP	DP	D	!!!	!
C	!!!	!!!	!!!	!!!
CP	!	!	!!!	!

Reg-Reg Arith (SUB only)
DEST: RD

B \ A	D	DP	C	CP
-	D	DP	!!!	CP
-	D	DP	!!!	CP
-	D	DP	!!!	CP
-	D	DP	!!!	CP

Immed Arith
DEST: RD

B \ A	D	DP	C	CP
D	!	!	!!!	!
DP	!	!	!!!	!
C	!!!	!!!	!!!	!!!
CP	!	!	!!!	!

Any Arith (Overflow)
DEST: RD

How do we generate tags?

We can use LLVM to statically analyze the codes, determine data types and generate tags for each 32/64 bit.

Out of scope for this project

Architectural Design

Extend register files

Tag determination in ALU

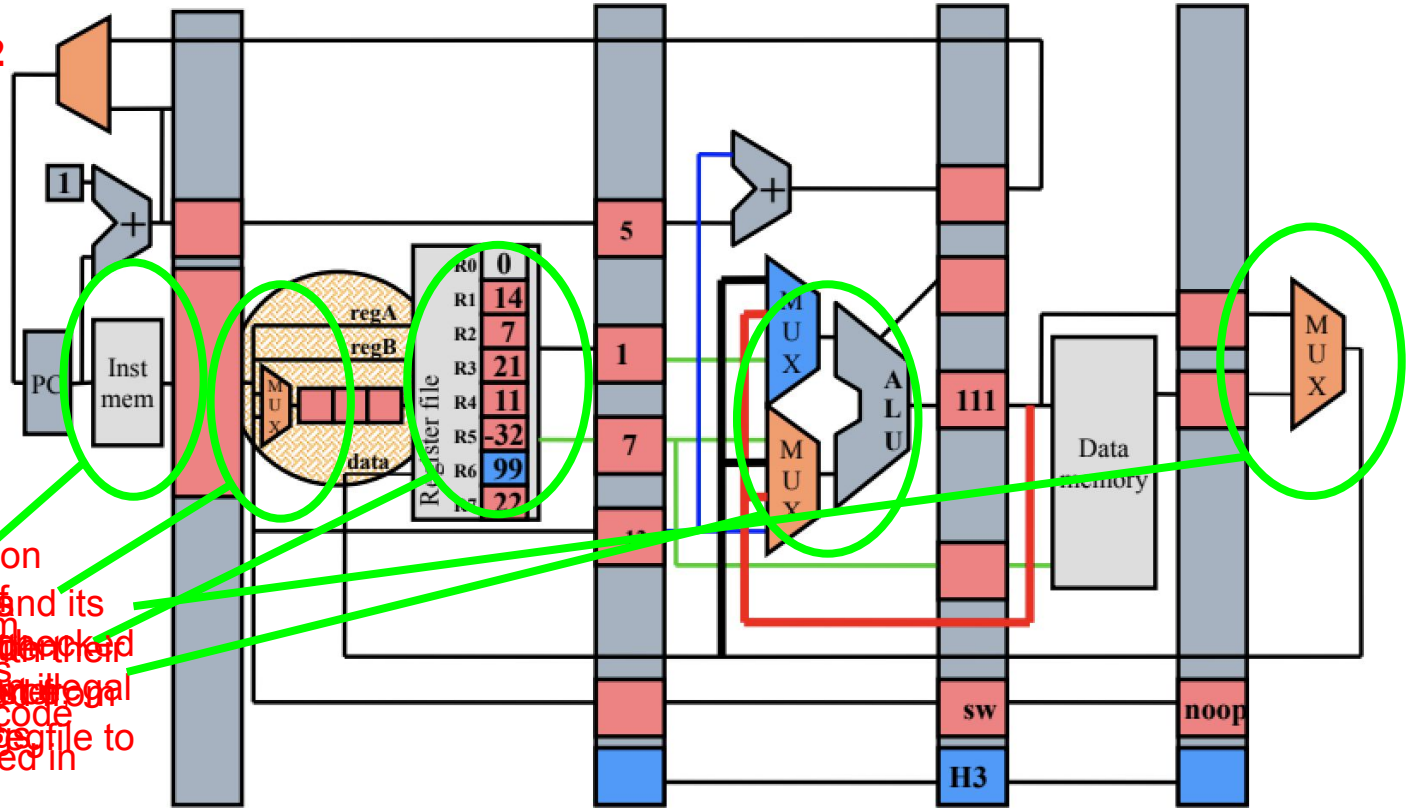
Tagging in writeback and forwarding logic

Tagging in memory hierarchy

Raise exception when rules violated

New instructions for debugging and flexibility

ADD rd,rs1,rs2

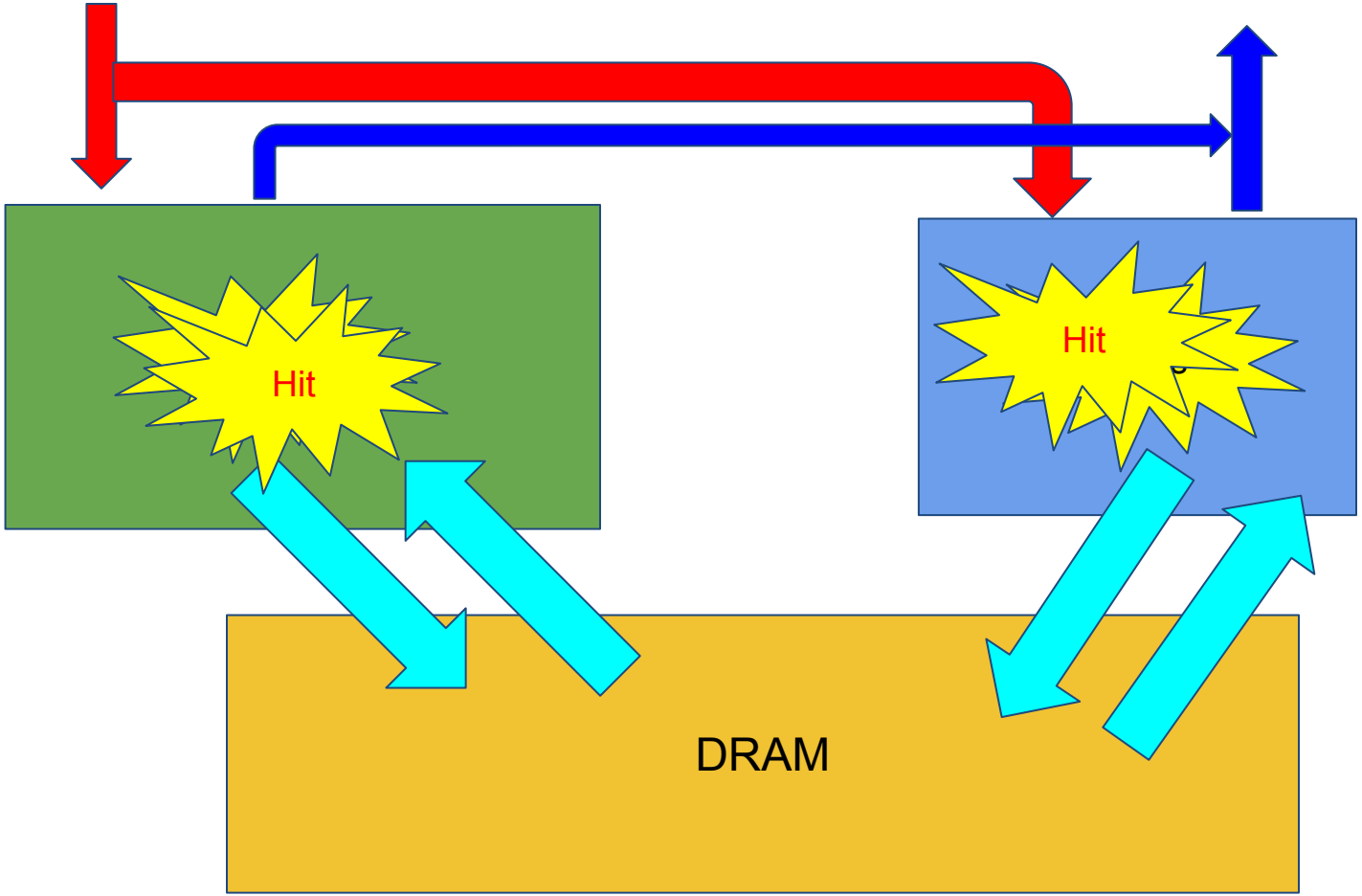


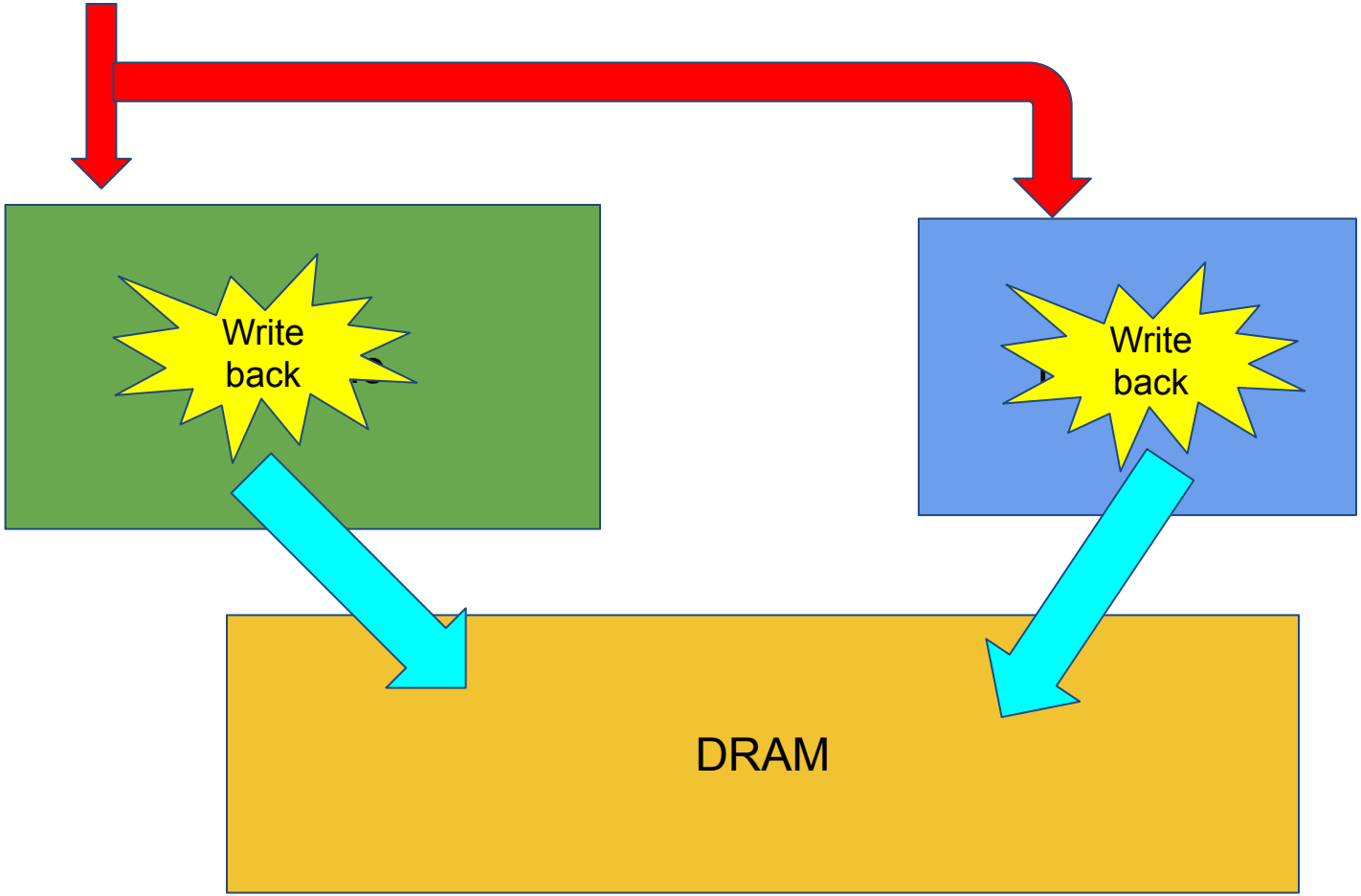
Before operation
Instruction
fetched from
Tag is
Verified as code
as it is placed in
Icache
exception is thrown.
Tag of dest reg is now
determined.

Memory Hierarchy Tag Propagation

DRAM: Tags are stored in a space in DRAM that is *non-addressable by software*. In this partition, tags are stored one after the other. This means a typical DRAM block of 64B can store 128 tags (at 4-bit tags).

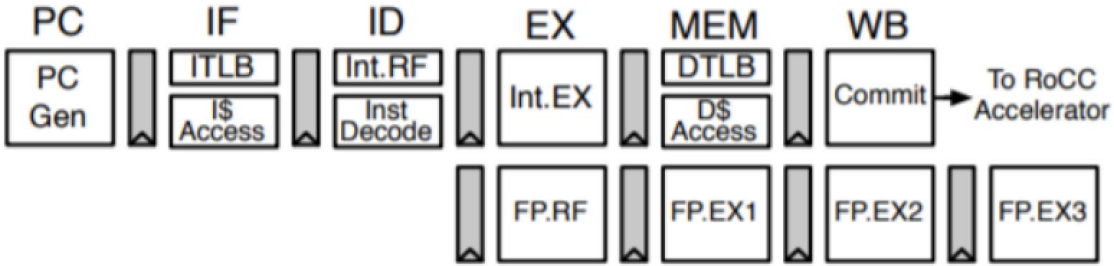
Tag Cache: When reading from the DRAM, we also issue a request to fetch the corresponding tags for that block. Since reading that tag block will also contain tags *irrelevant to the first DRAM request*, we cache the extraneous tags in case spatial locality is realized in subsequent accesses.
When write back to the DRAM, both data and tags will be written back.


Hit
Hit
DRAM


Write back
Write back
DRAM

Technical Details

- Rocket Core
  - Open-source: Allows us to make add-ons without having to reinvent the base architecture
  - Simple: 5-stage pipeline makes debugging and analysis simpler
  - Parameterized: Chisel allows us to easily change hardware module configuration to meet our needs



https://www.lowrisc.org/docs/tagged-memory-v0.1/rocket-core/

New Instructions

SETTAG $r1, CP Tag_Reg[1] = CP

CMPTAG $1, CP If(Tag_Reg[1] != CP) Raise Exception

Requirements:

1. No collision with other instructions.
2. Fit in 5 stage pipeline.
3. Handle hazards and bypass.

Trick:

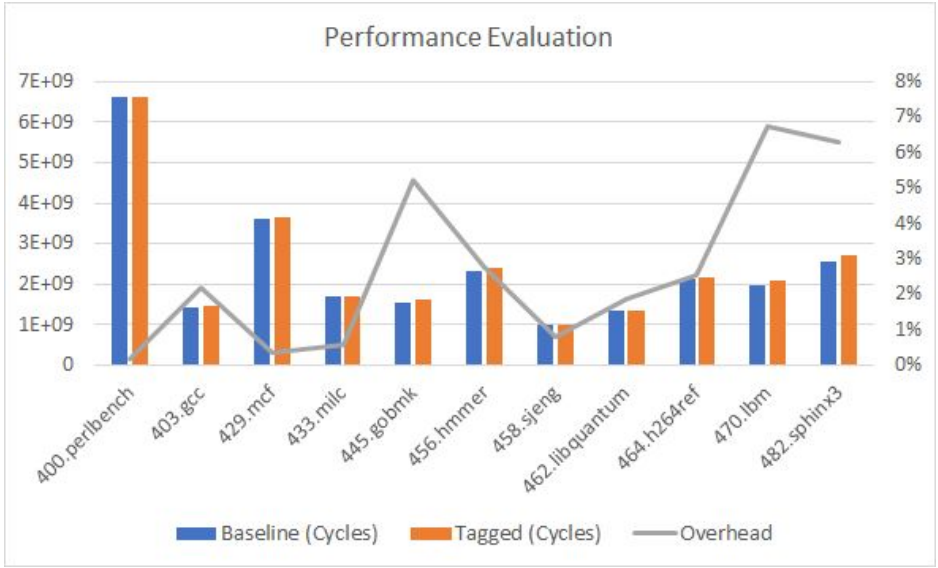
Format and decode the instruction similar to ADD $1, $0, $1.

Then the original rocket pipeline will handle bypass logic for us.

Analysis: Performance

Gem5 Simulation

8 kB icache
32 kB dcache
2 kB tag cache
1 GHz
No L2 cache
64 bit in order core



3% Performance Overhead on Average

Analysis: Area

- Synthesis Configuration

4 kB icache
16 kB dcache
2 kB tag cache
No floating point unit
32 bit 5 stage in order pipeline

- Resulting in ~0.35% area overhead

Discussion & Limitations

Discussion:

- Using larger tag cache or multi-layer cache hierarchy will likely decrease tag miss rate and thus performance overhead.
- More fine-grained rules and domain categories will lead to higher security and less false positives.

Limitations:

- Didn’t fully verify the design and implementation.
- Area evaluation is not on a full implementation, so the area overhead will only go up.

Conclusions

- We implemented architectural support to include tag propagation and rule checking in the rocket core pipeline.
- We added new instructions for the debugging purpose and also gives programmers more flexibility.
- We evaluated the full-scale system by simulation and synthesis. The results show that such design has low performance and area overhead.

Q&A