Semiconductor Research Corporation

Core Fuzzing - A Versatile Security Verification Platform

Alenkruth Krishnan Murali, Ashish Venkat
University of Virginia

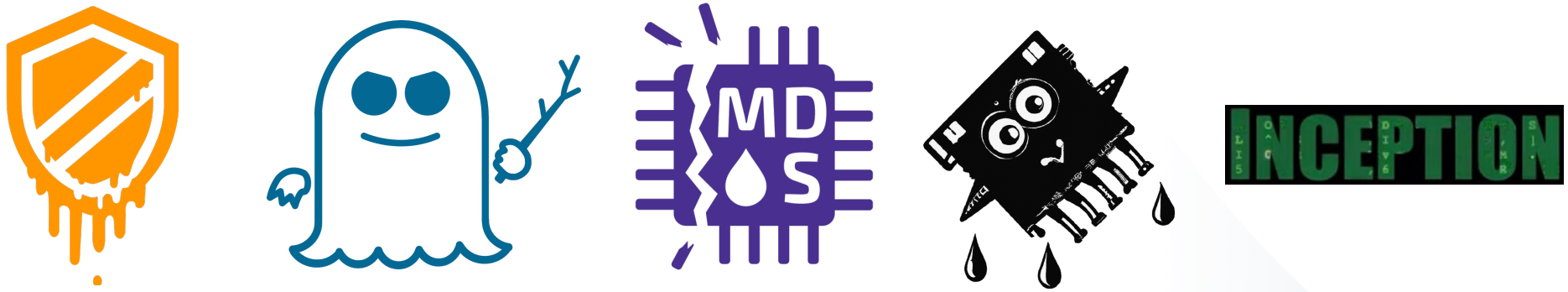
CADT-Task #3105.001: ProxyVM - A Scalable and Retargetable Compiler Framework for Privacy-Aware Proxy Workload Generation

Outline

- Motivation and Key Idea
  - Need for Security Verification
  - Current Security Verification Techniques
  - Core Fuzzing - Key Idea
  - Thought Experiments
- Overview of Core Fuzzing
- Preliminary Results
- Conclusions

Need for Security Verification

- Modern architectures are complex
- Verification time increases, effort increases
- Endless stream of hardware and software attacks



- Robust and reliable verification techniques are necessary

Current Security Verification Techniques

- Formal verification based approaches
  - Static/Compile time methods
  - Verification time ∝ Design/Software complexity
- Hardware security verification
  - Different abstraction levels - software to gates
  - Convert design to (formal) verification tool language - additional step
- Black-Box CPU verification
  - Directed test generation through fuzzing
  - Needs a formal specification of the ISA - additional step

Core Fuzzing - Key Idea

- Core fuzzing is a flexible, fast, and automatic security verification framework.
- Key idea - Fuzz the microarchitecture of the processor during runtime while keeping the test program constant.
- Based on software fuzzing.
- Fuzzing during runtime exposes previously unseen execution paths and microarchitectural side-effects.
- Identify vulnerabilities in SW and HW by monitoring information flow.

Core Fuzzing - Thought Experiment 1

Experiment 1:

- Branch is not immediately followed by load
- Machine with unlimited execution units and 100 entry ROB
- Due to a smaller window, the load is never speculatively executed
- Increase ROB size to 250.
  - Increases the number of in-flight instructions
  - Upon misspeculation, the branch + speculated load act as a spectre gadget leaking information through a side channel (cache)

Spectre PoC code:

```
if (index < array_size){
  // 200 ALU operations without  loads/stores
  dummy1 = array2[array1[index]];
}
```

- Varying ROB size exposes bug in software exploiting the hardware
- Not apparent during normal execution

Core Fuzzing - Thought Experiment 2

Experiment 2:

- Test program containing a secret dependent operation running on a machine with 10 multipliers
- Reduce the number of multiplier ports to 1
  - Secret is leaked implicitly through the execution time of instruction C.
  - Such scenarios can occur
    - In a SMT machine
    - Resource constrained devices

```
if (secret == “hello”){
  result = a * b; (A)
}
else (secret == “world”){
  result = a / b; (B)
}
dummy = a * b; (C)
```

- Cannot be noticed unless tested on diverse microarchitectures

Core Fuzzing - Thought Experiment 3

Experiment 3:

0000000080002a10 <secret>:

80002a10: 2790

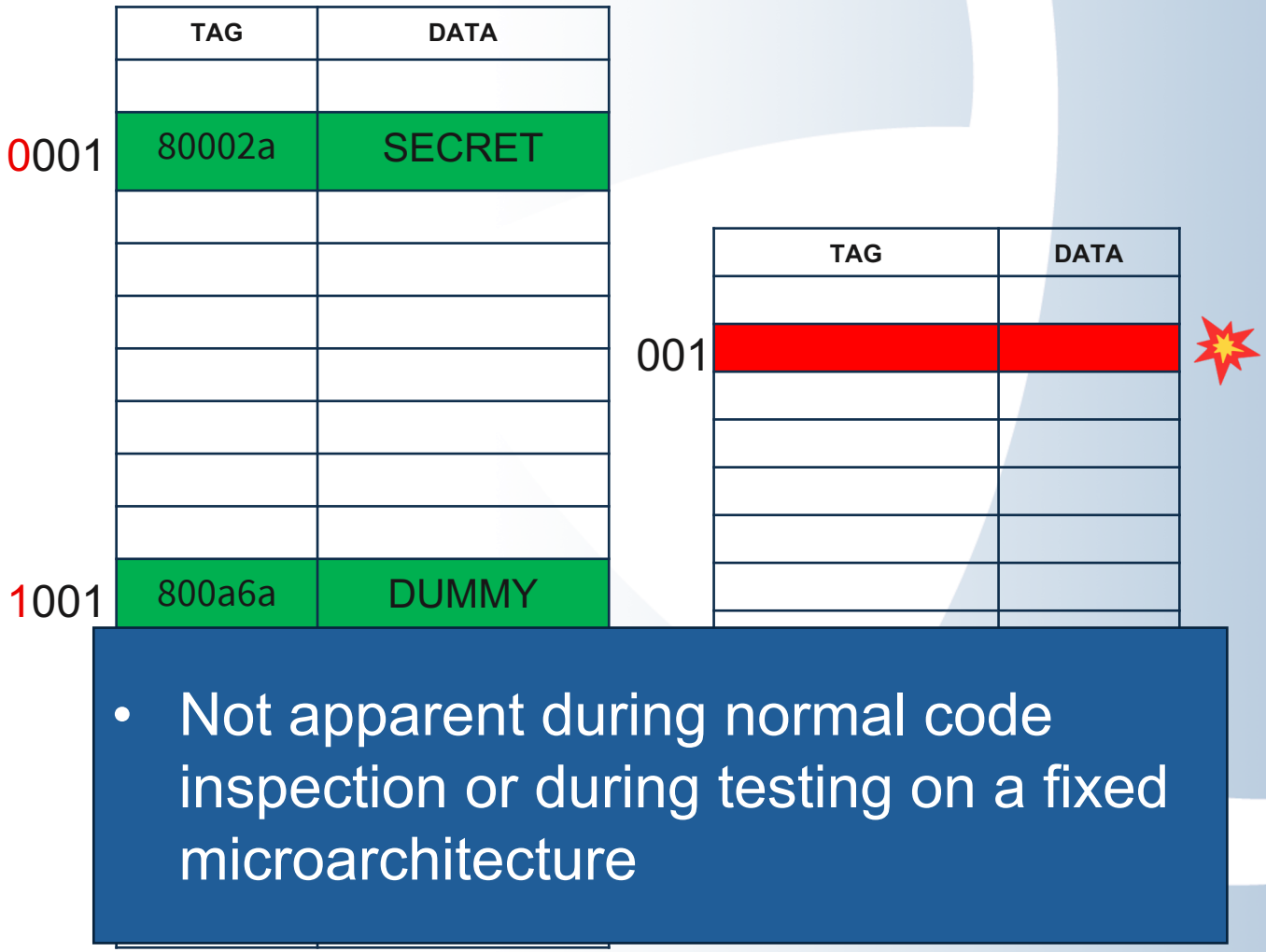
80002a12: 8000

…

00000000800a6a90 <dummy>:

800a6a90: 8000

- Reduce the number of sets from 16 to 8.
  - Secret data and attacker data contend for the same cache line.
  - Resource contention can be used to deny service.
  - Branch History Tables are similarly vulnerable.



- Not apparent during normal code inspection or during testing on a fixed microarchitecture

Why is Core Fuzzing Better?

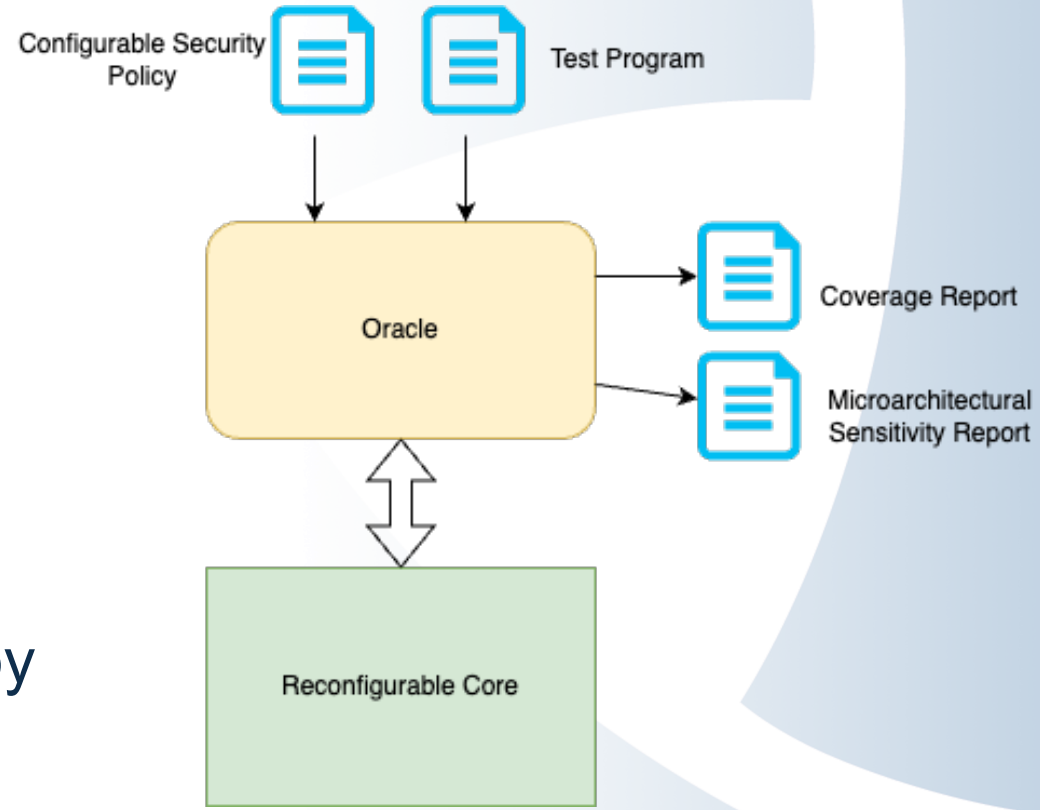
- Given a design specification, the tool automatically fuzzes to find a representative sample of possible *valid* microarchitectural configurations
  - Invalid configurations
    - Cache line with odd number of ways, unpipelined processor, static branch prediction, …
  - Security-aware design space exploration
- Hardware-based verification tool - faster than static methods
- Software bugs and hardware bugs are exposed
  - Vulnerabilities in microarchitectural configurations
  - Vulnerabilities in software that lead to security policy violation

Outline

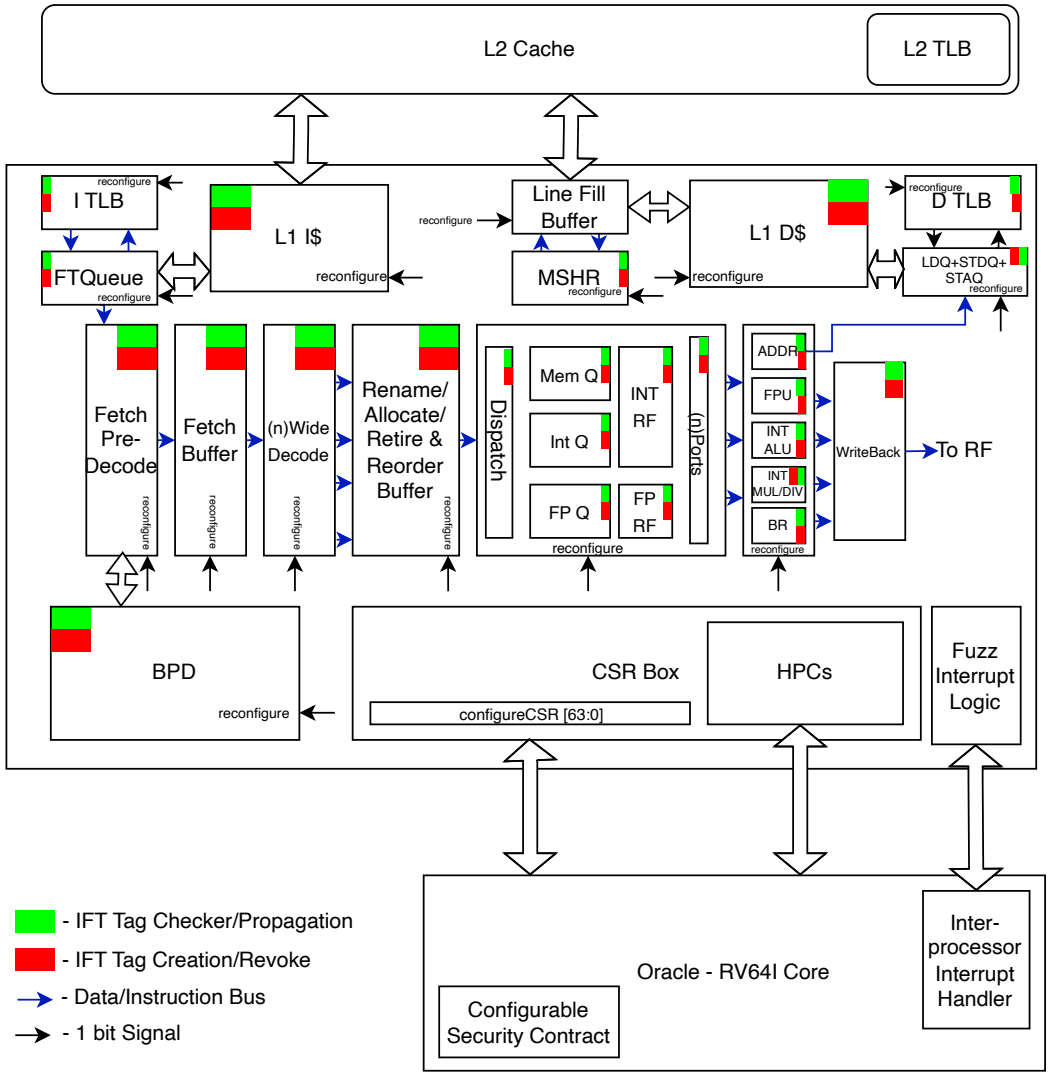
- Motivation and Key Idea
- Overview of Core Fuzzing
  - Components of the Framework
  - Reconfigurable core
  - Dynamic Information Flow Tracking(DIFT) in the reconfigurable core
  - Oracle
  - Methodology
- Preliminary Results
- Conclusions

Core Fuzzing - Framework

- Three components
  - Reconfigurable core
  - Oracle
  - Configurable security policy
- Oracle and reconfigurable core in master/subordinate arrangement
- Reconfigurable RISC-V BOOM Core
  - Implements Dynamic Information Flow Tracking (DIFT) at module interfaces
- Oracle monitors execution on reconfigurable core and implements the fuzzer
- The configurable security policy is provided by the user
  - List of acceptable violations
  - List of acceptable leakage channels



Core Fuzzing - Reconfigurable Core

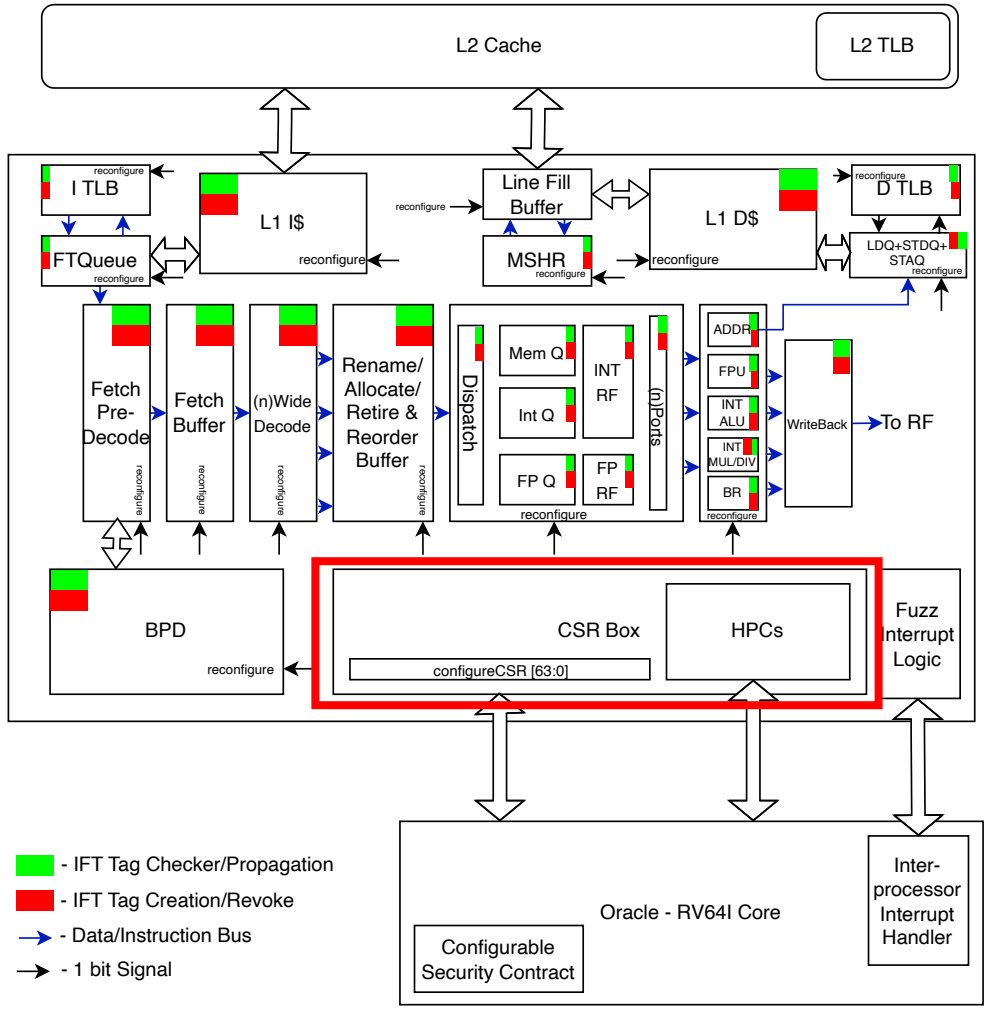


Pipeline of the reconfigurable BOOM core

- Red - DIFT tag creation and revoke
- Green - DIFT tag checker and propagation
- Modules with DIFT units are reconfigurable
- Oracle - interfaced with the reconfigurable core

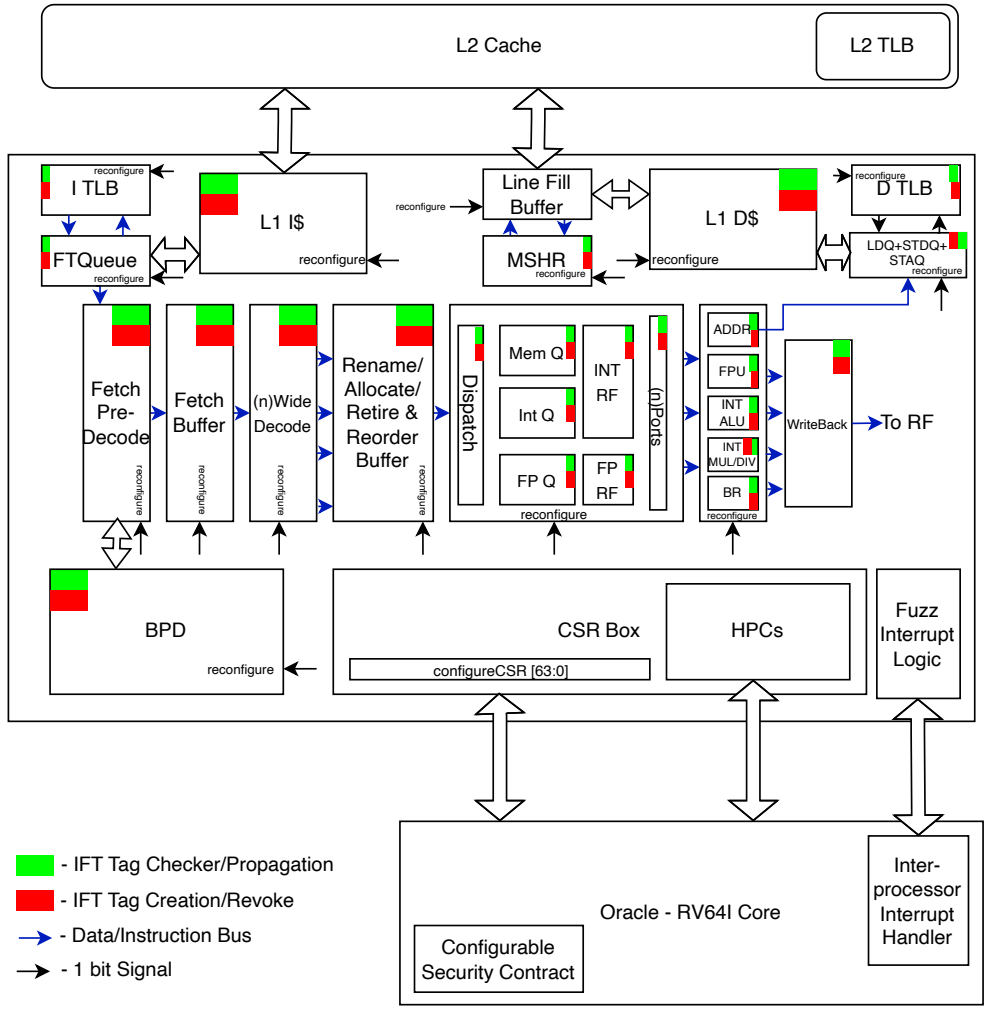
https://boom-core.org

Core Fuzzing - Reconfigurable Core



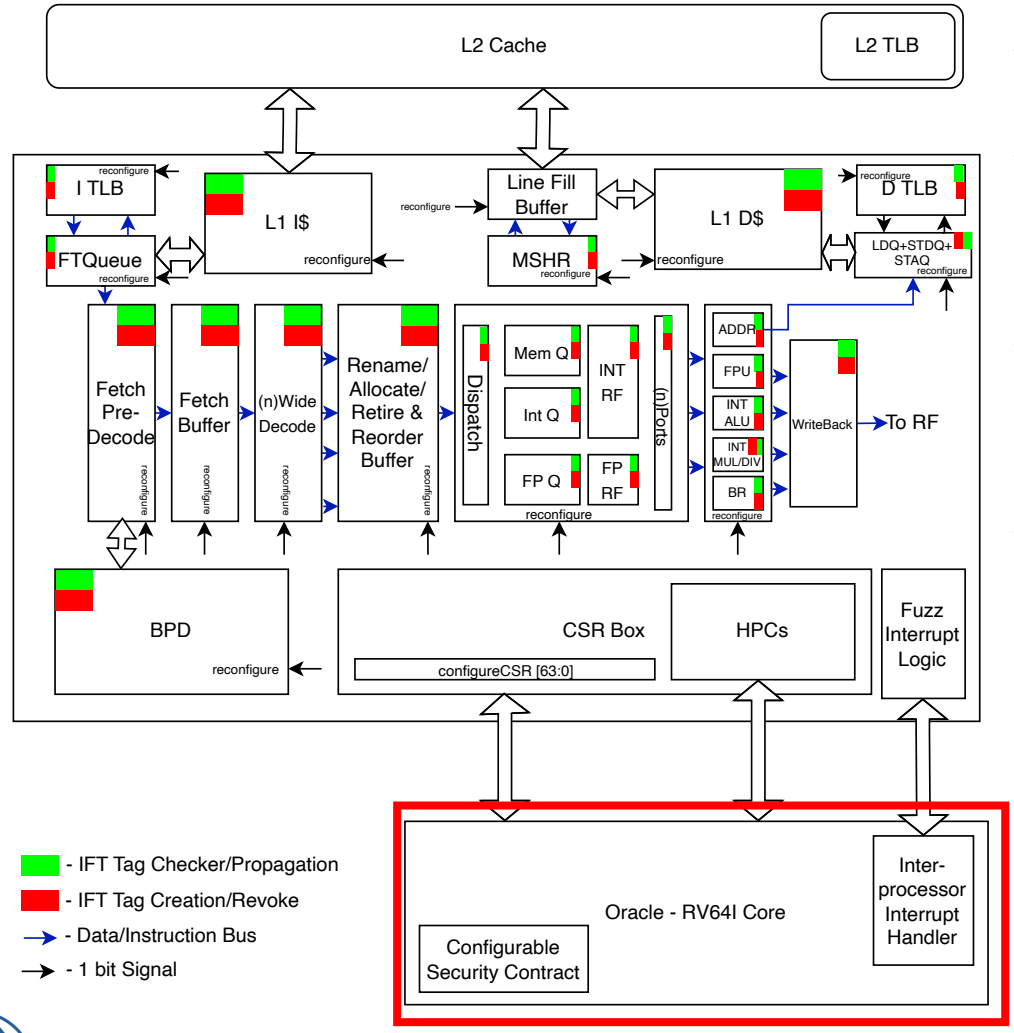
- Configurable modules
  - Branch predictor, instruction window, execution ports, issue width, cache organization….
- Reconfigurations
  - triggered by the oracle
  - dedicated custom CSR(s)
- DIFT units enforce the configurable security contract
- The reconfigurable core populates custom CSRs with information flow for the oracle to monitor

Core Fuzzing - DIFT



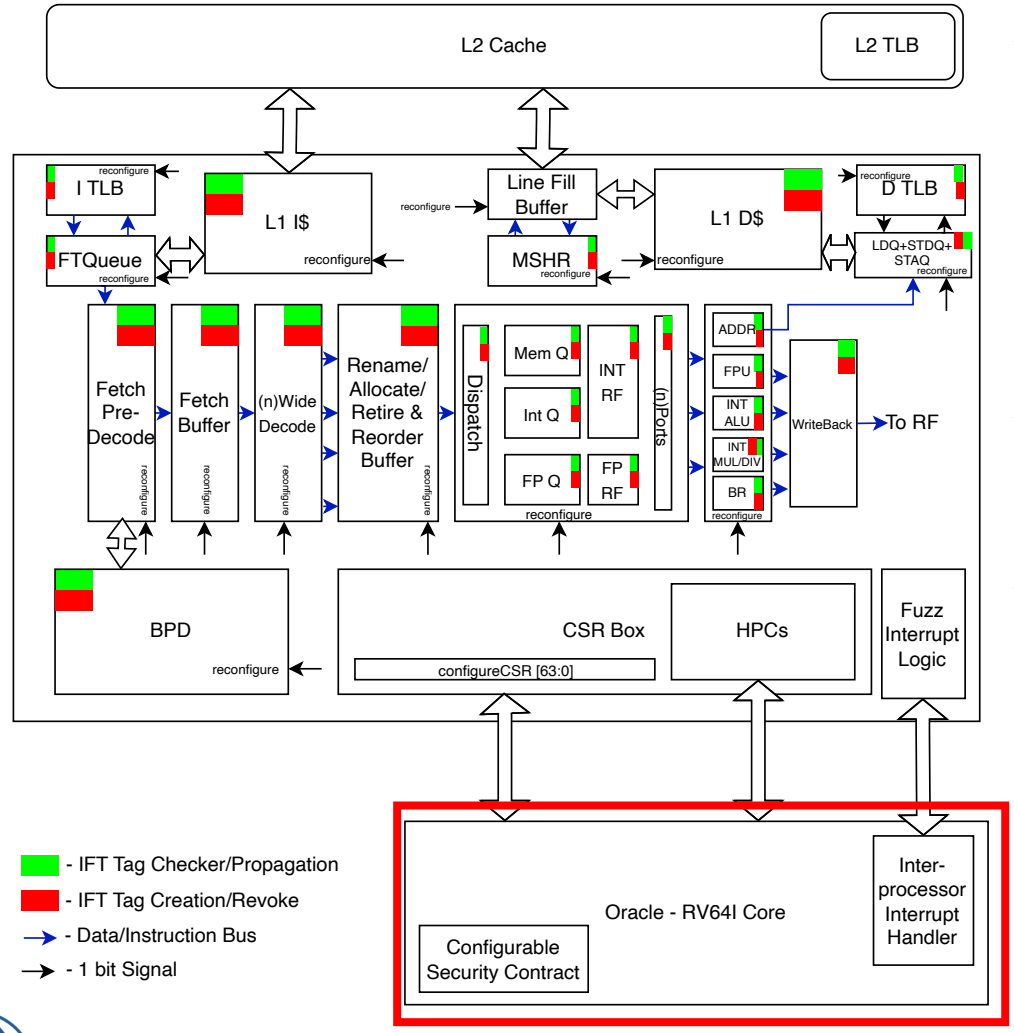
- Initial tagging - Red
  - Data/Instruction entering the caches
- Propagation and checking units - Green
  - Check interaction of data with different tags
  - Propagates the tags based on interactions
- Tag revoke - Red
  - Instruction retirement

Core Fuzzing - Oracle



- Brain of the Core Fuzzing framework
- RISC-V core supporting privileged ISA and interprocessor interrupts
- Sets up the test program and the reconfigurable core
- The fuzzer provides a initial list of microarchitectural configurations for a given design specification

Core Fuzzing - Oracle



- Continuously probes the custom CSRs and Hardware Performance Counters (HPCs)
  - Uses probed values to trigger reconfigurations
- Fuzzer uses probed values to decide the next configuration
- End of the run
  - Report with SW and HW bugs that led to policy violation
- Details in the paper

Core Fuzzing - Methodology.

- Built with Open-Source tools
- Reconfigurable core - **B**erkeley **O**ut-of-**O**rder **M**achine
- Chipyard framework - SoC generation and Verilator simulations
- Firesim framework - deployment on AMD-Xilinx U250 FPGAs

1. https://boom-core.org
2. https://github.com/ucb-bar/chipyard
3. https://fires.im
4. https://riscv.org

Outline

- Motivation
- Overview of Core Fuzzing
- Preliminary Results
  - Spectre-v1 PoC with BPU reconfiguration
- Conclusions

Spectre-v1 and Branch Predictor Reconfiguration

- Spectre-v1 PoC code tuned to mistrain the Gshare predictor.
- BOOM v3 uses a TAGE branch predictor.
- During the fuzzing run, the core reconfigures to use a Gshare predictor in place of the TAGE predictor.

Secret Value

```
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 45151
m[0x0x80002790] = want(") =?= guess(hits,dec,char) 1.(9, 34, ") 2.(1, 1, )
m[0x0x80002791] = want(S) =?= guess(hits,dec,char) 1.(1, 1, ) 2.(1, 2, )
m[0x0x80002792] = want(e) =?= guess(hits,dec,char) 1.(1, 1, ) 2.(1, 2, )

Triggering a reconfiguration
m[0x0x80002793] = want(c) =?= guess(hits,dec,char) 1.(6, 99, c) 2.(1, 1, )
m[0x0x80002794] = want(r) =?= guess(hits,dec,char) 1.(7, 114, r) 2.(1, 1, )
m[0x0x80002795] = want(e) =?= guess(hits,dec,char) 1.(8, 101, e) 2.(1, 1, )
m[0x0x80002796] = want(t) =?= guess(hits,dec,char) 1.(7, 116, t) 2.(1, 1, )
*** PASSED *** Completed after 14463565 cycles
[UART] UART0 is here (stdin/stdout).
```

Guessed value (extracted through a side channel)

Spectre-v1 and Branch Predictor Reconfiguration

- Spectre-v1 PoC code tuned to mistrain the Gshare predictor.
- BOOM v3 uses a TAGE branch predictor.
- During the fuzzing run, the core reconfigures to use a Gshare predictor in place of the TAGE predictor.

TAGE BPU

Gshare BPU

```
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 45151
m[0x0x80002790] = want(") =?= guess(hits,dec,char) 1.(9, 34, ") 2.(1, 1, )
m[0x0x80002791] = want(S) =?= guess(hits,dec,char) 1.(1, 1, ) 2.(1, 2, )
m[0x0x80002792] = want(e) =?= guess(hits,dec,char) 1.(1, 1, ) 2.(1, 2, )

Triggering a reconfiguration
m[0x0x80002793] = want(c) =?= guess(hits,dec,char) 1.(6, 99, c) 2.(1, 1, )
m[0x0x80002794] = want(r) =?= guess(hits,dec,char) 1.(7, 114, r) 2.(1, 1, )
m[0x0x80002795] = want(e) =?= guess(hits,dec,char) 1.(8, 101, e) 2.(1, 1, )
m[0x0x80002796] = want(t) =?= guess(hits,dec,char) 1.(7, 116, t) 2.(1, 1, )
*** PASSED *** Completed after 14463565 cycles
[UART] UART0 is here (stdin/stdout).
```

Outline

- Motivation
- Overview of Core Fuzzing
- Preliminary Results
- Conclusions

Conclusions

- Core Fuzzing is a quick, flexible, and automatic security verification solution
  - Hardware-based - faster than static techniques
  - User defined configurable security policy
  - Automatic guided fuzzing
- Reconfiguration during runtime
  - Expose software and hardware bugs not visible during normal execution
  - Security-aware design space exploration

Technology Transfer

- New Idea - Few months old - First Public presentation.
- No industry interactions as of date.
  - Interested? Reach out to us.

Thank you. Questions?

alenkruth@virginia.edu

Backup slides

Information Flow Tracking

- Partial DIFT support in the reconfigurable core.
- Tagging happens in DCache during write.
- Tags are propagated to the Load Queue

```
[dcache] Tag of current Dcache request is 1

[lsu][ldq] Tag of the current ld is - 1


[dcache] Tag of current Dcache request is 0

[lsu][ldq] Tag of the current ld is - 0
3 0x00000000800012cc (0x18e7f7d3) f15 0xffffffff43000000
```

- Working on full IFT mechanism with ability to populate custom CSRs.