IntroSpectre: A Pre-Silicon Framework for Discovery and Analysis of Transient Execution Vulnerabilities

Moein Ghaniyoun[†], Kristin Barber[†], Yinqian Zhang[§], Radu Teodorescu[†]

Department of Computer Science and Engineering

May 2022

RISC-V
uSC SIG

[†] The Ohio State University

[§] SUSTech
Southern University of Science and Technology

COMPUTER ARCHITECTURE RESEARCH LAB

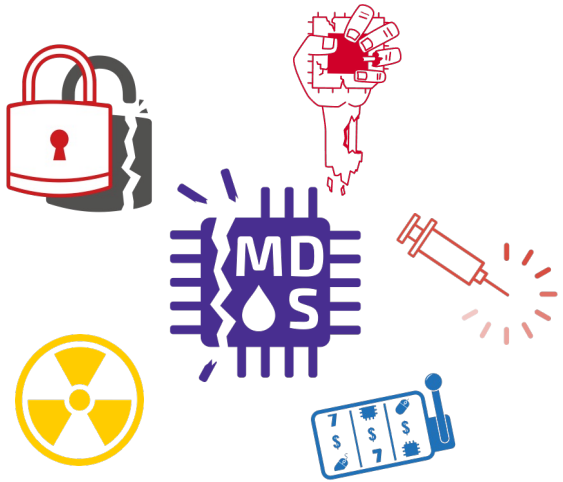
http://arch.cse.ohio-state.edu

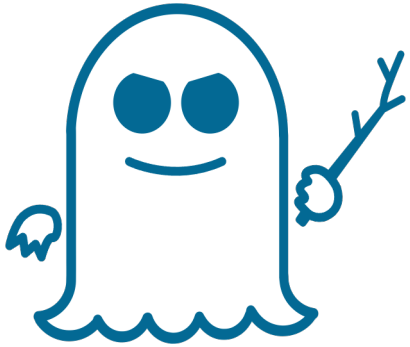
Background

- ***Speculative execution*** has become a must-have feature for almost all mid-high range processors
- Performance optimization ≈ "*Potential*" security flaw ?
- Recent works demonstrate the existence of a vast attack surface on speculative execution

Background

- The discovery of ***Meltdown*** and ***Spectre*** in 2018 has led to an explosion of transient execution attacks

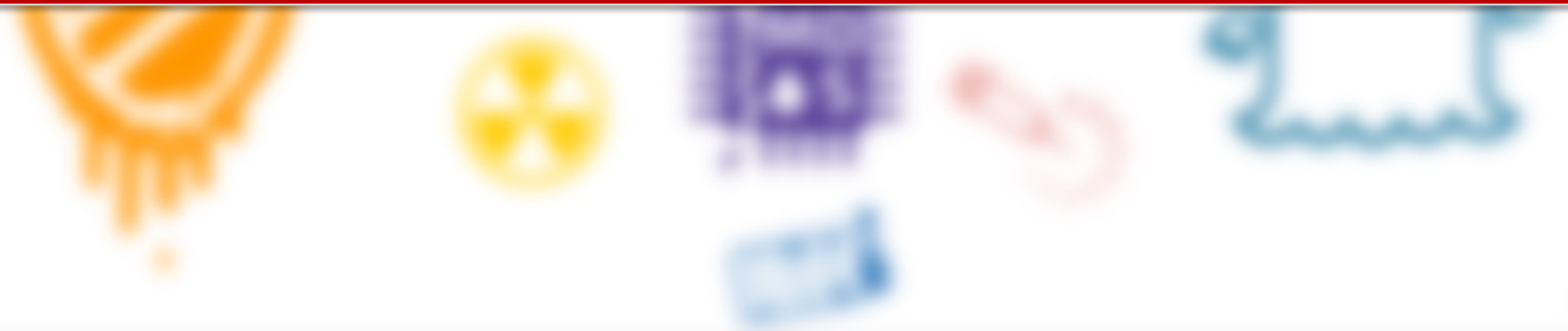




Background

- The discovery of Meltdown and Spectre in 2018 has led to

There has been significant efforts in both industry and academia to ***mitigate*** and/or ***detect*** transient execution attacks



Background

Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis

Daniel Moghimi, *Worcester Po*
Graz University of Technology; Berk S
Michael Schwarz, *Graz*

https://www.usenix.org/conference/useni

DIFFUZZ: Differential F Ana

Shirin Nilizadeh*
University of Texas at Arlington
Arlington, TX, USA
shirin.nilizadeh@uta.edu

Yannic Noll
Humboldt-Universitä
Berlin, Germa
yannic.noller@hu-

***Abstract*—Side-channel attacks allow an adversary to uncover secret program data by observing the behavior of a program with respect to a resource, such as execution time, consumed memory or response size. Side-channel vulnerabilities are difficult to reason about as they involve analyzing the correlations between resource usage over multiple program paths. We present DIFFUZZ, a fuzzing-based approach for detecting side-channel vulnerabilities related to time and space. DIFFUZZ automatically detects these vulnerabilities by analyzing two versions of the program and using resource-guided heuristics to find inputs that *maximize* the difference in resource consumption between secret-dependent paths. The methodology of DIFFUZZ is general and can be applied to programs written in any language. For this paper, we present an implementation that targets analysis of JAVA programs, and uses and extends the KELINCI and AFL fuzzers. We evaluate DIFFUZZ on a large number of JAVA programs and demonstrate that it can reveal unknown side-channel vulnerabilities in popular applications. We also show that DIFFUZZ compares favorably against BLAZER and THEMIS, two state-of-the-art analysis tools for finding side-channels in JAVA programs.**

***Index Terms*—vulnerability detection; side-channel analysis;**

SPECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities

Yuan Xiao
The Ohio State University
xiao.465@osu.edu

Yinqian Zhang
The Ohio State University
yinqian@cse.ohio-state.edu

Radu Teodorescu
The Ohio State University
teodores@cse.ohio-state.edu

***Abstract*—SPEculative Execution side Channel Hardware (SPEECH) Vulnerabilities have enabled the notorious Meltdown, Spectre, and L1 terminal fault (L1TF) attacks. While a number of studies have reported different variants of SPEECH vulnerabilities, they are still not well understood. This is primarily due to the lack of information about microprocessor implementation details that impact the timing and order of various micro-architectural events. Moreover, to date, there is no systematic approach to quantitatively measure SPEECH vulnerabilities on commodity processors.**

This paper introduces SPEECHMINER, a software framework for exploring and measuring SPEECH vulnerabilities in an automated manner. SPEECHMINER empirically establishes the link between a novel two-phase fault handling model and the exploitability and speculation windows of SPEECH vulnerabilities. It enables testing of a comprehensive list of exception-triggering instructions under the same software framework, which leverages covert-channel techniques and differential tests to gain visibility into the micro-architectural state changes. We evaluated SPEECHMINER on 9 different processor types, examined 21 potential vulnerability variants, confirmed various known attacks, and

software [27], [21], [3], [25]. Moreover, new variants of these vulnerabilities are constantly being discovered by hackers and security researchers. Prominent examples include LazyFP [32], Meltdown-RW [16], Fallout [29], ZombieLoad [30], *etc*.

A major challenge faced by researchers, software developers and hardware designers is the ignorance about the fundamental question of what determines the success or failure of an attack. Without a concrete general conclusion over the nature of these attacks, great efforts are put into figuring out unique mitigation for each newly-emerging variant. Evaluating variants is also difficult with only random attempts of seemingly relative implementation tricks, hoping for a successful exploitation. Three aspects of complexity lead to the difficulty for a general conclusion to be made. First, the attacks vary greatly from each other. They have different threat models and exploit different instructions. And not enough details are provided about their implementation. Second, the micro-architectural states during execution is unobservable and unpredictable. The aggressive speculative and out-of-order pro-

SpecFuzz: Bringing Spectre-type vulnerabilities to the surface

ɔ and Bohdan Trach, *TU Dresden;*
chnion; Christof Fetzer, *TU Dresden*

rence/usenixsecurity20/presentation/oleksenko

Blackbox Side-channel ty Microarchitectures

urth*, Herbert Bos*, and Kaveh Razavi*
†Intel Corporation

cryptographic keys) by examining changes made by a victim's execution to the state of shared microarchitectural components such as caches [1, 2, 3, 4, 5], cache directories [6], TLBs [7], and branch predictors [8, 9]. Such attacks are typically based on *whitebox* side-channel analyses which require heroic reverse engineering efforts to gain a deep understanding of the target component and then craft component-specific exploitation primitives (e.g., the ability to track victim cache accesses by actively forcing evictions). Such manual efforts need to be repeated for each new component and each (micro)architecture, in search of new, dedicated exploitation primitives.

In this paper, we present ABSynthe, an automatic, *black box* approach towards synthesizing microarchitectural side channels in a more general and sustainable fashion, by exploiting contention on shared resources. Its blackbox analysis requires no reverse engineering effort on the part of the attacker, and eschews complicated eviction strategies that require deep

Transient Exe. Vuln. Detection

- Discovering all paths that *transient execution* may lead to secret data leakage is not straightforward
- Effects of **speculated** instructions are **rolled back** after the processor detects mis-speculation
- Creating a major **roadblock** to understanding the **negative** **side-effects** of transient execution

Motivation

- **Transient state** will **never** be **architecturally** visible if the executed instructions are on the **wrong** path

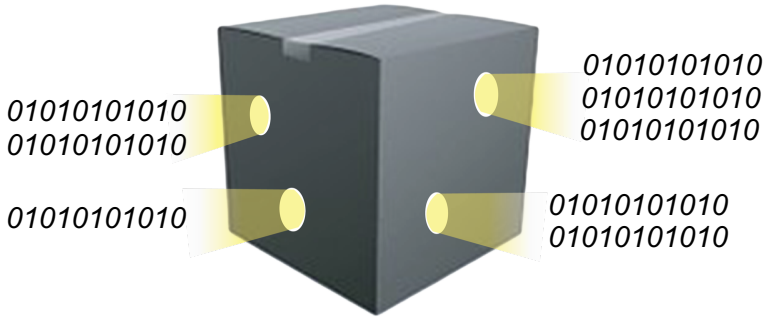
How to know what is inside this **closed** box?

- Many existing tools used for transient execution vulnerability ***detection*** rely on **covert/side channels**

Motivation

- Reliance on ***covert/side channels*** prevents a systematic evaluation of transient execution vulnerabilities
- Limits the **visibility** into potential leakage to only known channels

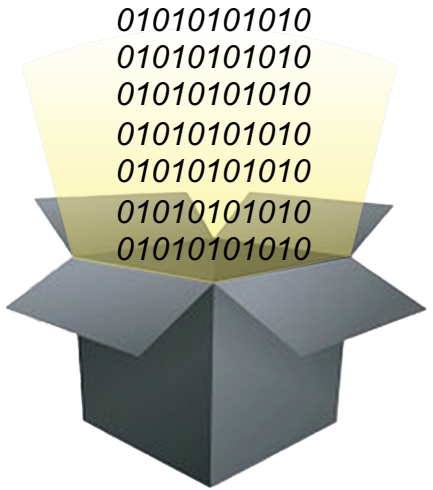
Guess what is in the **box** just by poking **holes**!



IntroSpectre

- ***IntroSpectre*** tackles the the problem of **visibility** into the internal state of the processor by integrating into **RTL verification flow**
- All relevant u-arch structures are monitored in the cycle-accurate simulation

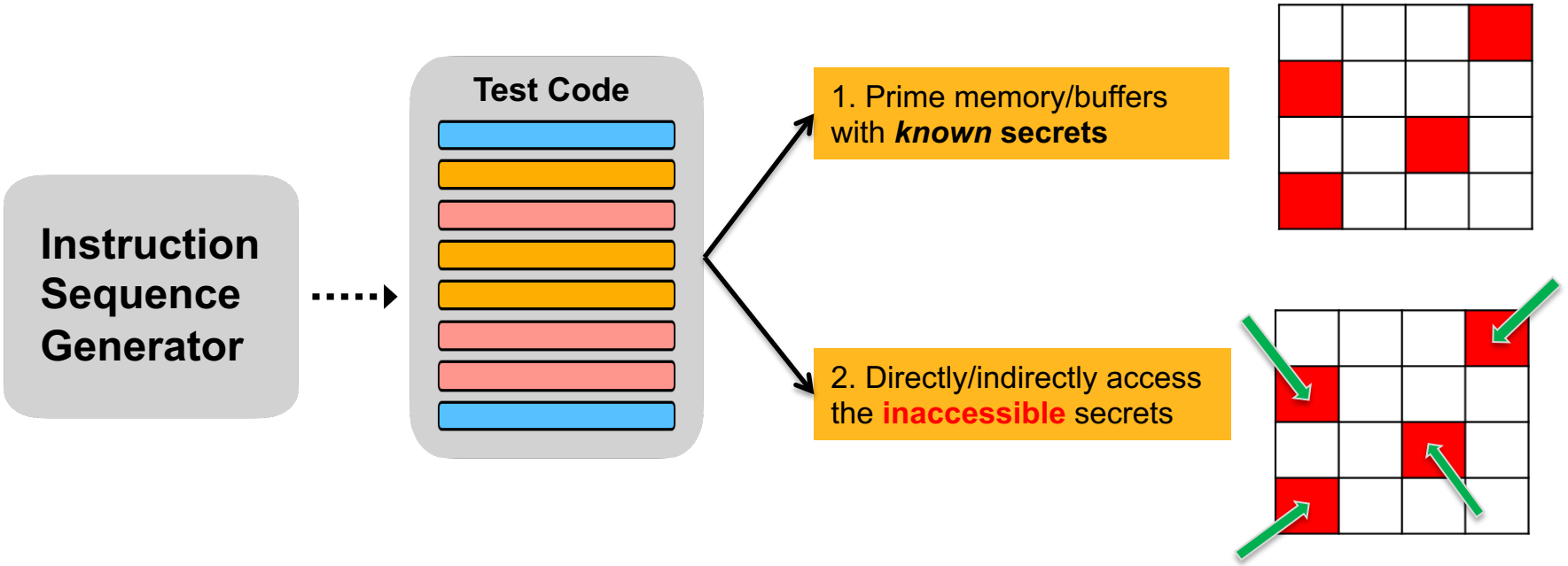
The box is **open** now! We can **see** what is inside!



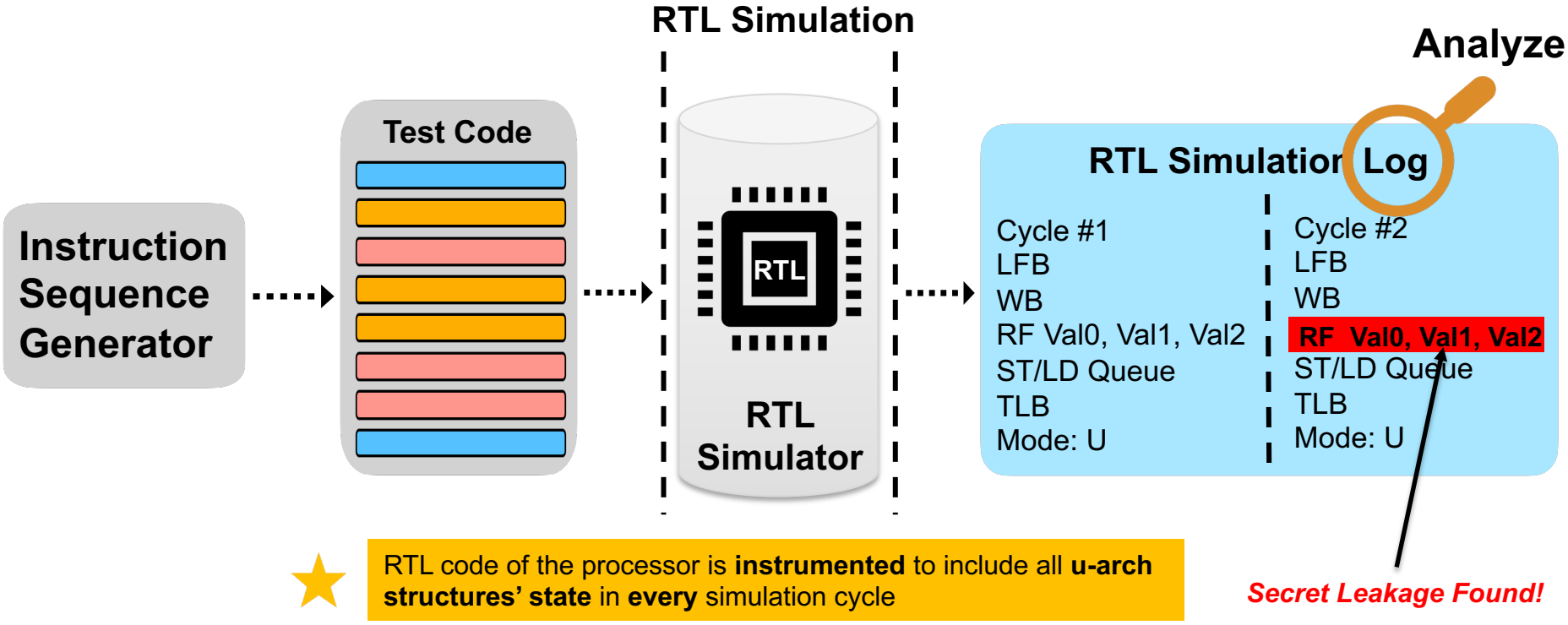
Outline

- IntroSpectre Framework
- Design
  - Instruction Sequence Generation
  - Leakage Analyzer
- Evaluation

IntroSpectre Framework



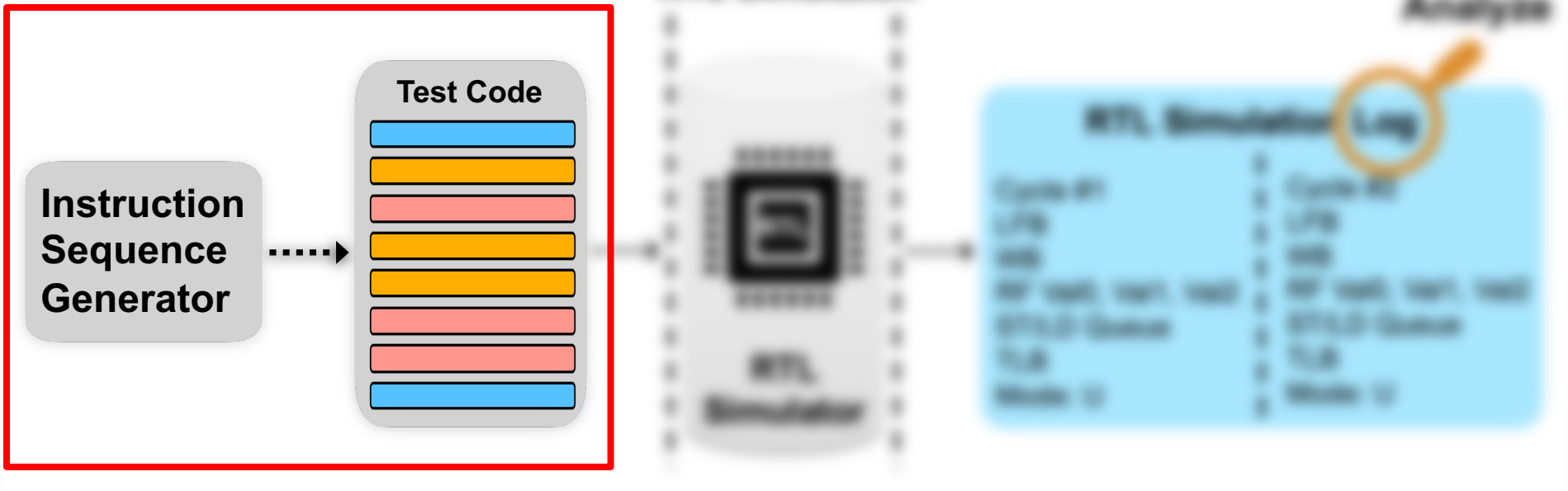
IntroSpectre Framework



Outline

- IntroSpectre Framework
- Design
  - Instruction Sequence Generation
  - Leakage Analyzer
- Evaluation

IntroSpectre Framework



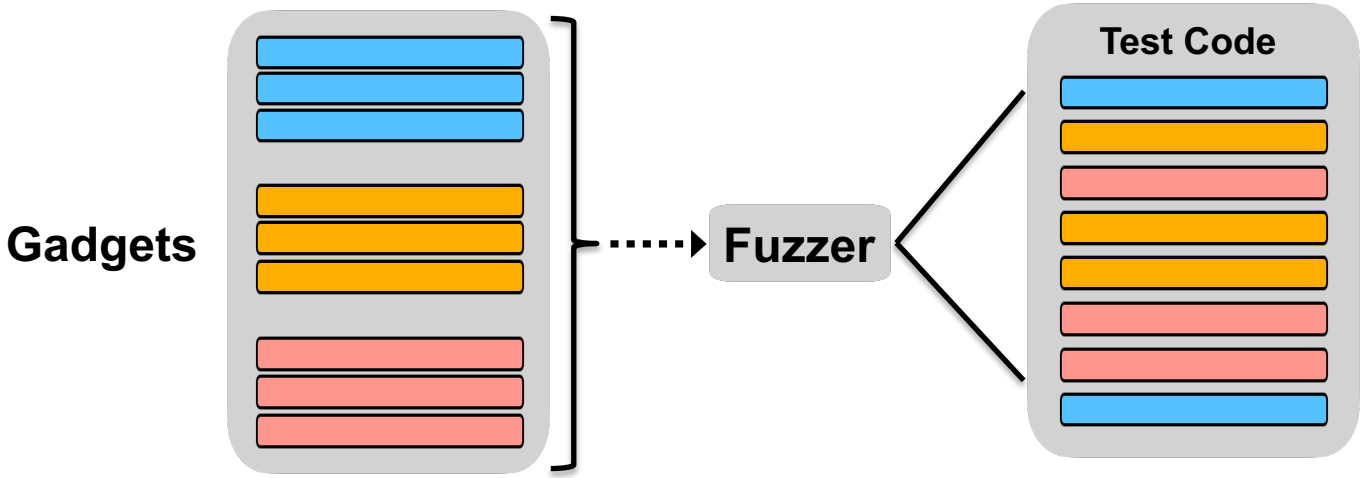
Instruction Sequence Generation

IntroSpectre Framework

How to access the inaccessible data in a comprehensive manner?

The Gadget Fuzzer

- ***IntroSpectre*** uses a set of **targeted gadgets** and employs **guided fuzzing** to select **relevant** combinations of these gadgets to generate **test cases** that are run through the RTL simulator
- The final output of the fuzzer includes instructions for both ***priming*** and ***accessing*** secret data



Gadget Types

Main Gadgets

The **core** of the leakage test sequence. They include speculation primitives and access instructions

Helper Gadgets

Setup Gadgets

Help the fuzzer to **establish the preferred** state for execution of main gadgets

Gadget Types

Multiple Gadget Variants

- The fuzzing is **not limited** to only combining gadgets! Most gadgets are **parametrized**, adding another dimension to the fuzzing space
- These **parameters** are **randomized** during instruction generation

Main Gadgets

- Many of them are crafted based on **known** transient execution attacks
- The rest target corner cases by employing **torturous** instructions

Main gadgets can benefit from certain u-arch states!

Example:

```
Meltdown_us()
    LOAD x10, [Supervisor Address]
```

Requirements:

Supervisor address should be present in L1D$

Main gadgets

- Many of them are crafted based on known transient execution attacks

How to fulfill the required micro-architectural state for main gadgets?

Example:

```
LOAD x1A, [Supervisor Address]
```

Requirements:

Supervisor address should be present in L1DS

Main gadgets

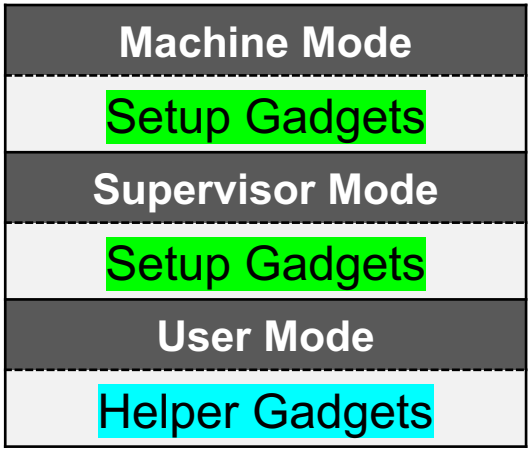
How to fulfill the required micro-architectural state for main gadgets?

Main gadgets come with state requirements!

Helper & Setup Gadgets!

Helper & Setup Gadgets

- These gadgets are ***selected*** by fuzzer based on the requirements of main gadgets
- The only difference between helper and setup gadgets is their **execution privilege**

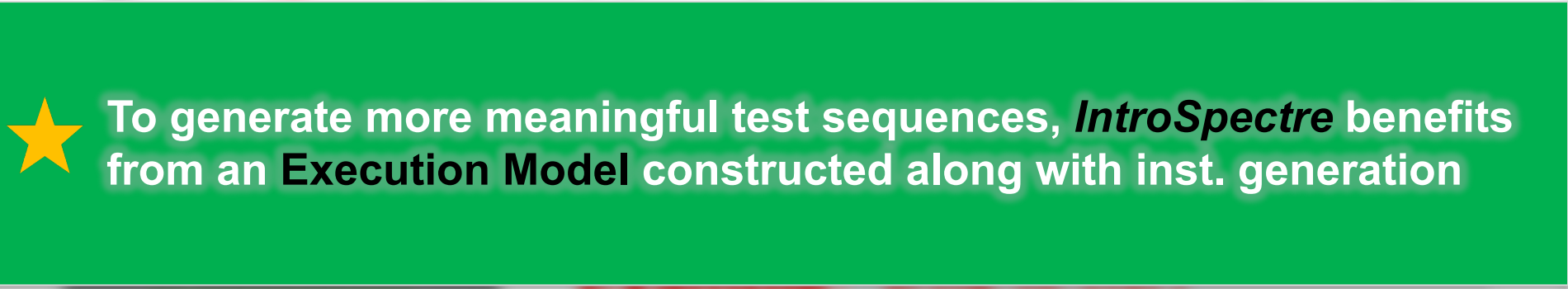


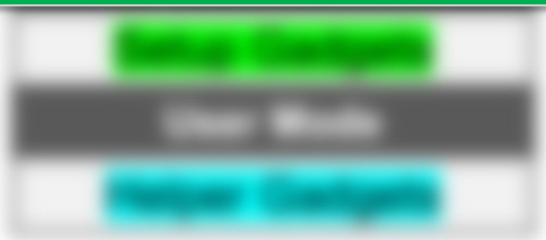
Example:

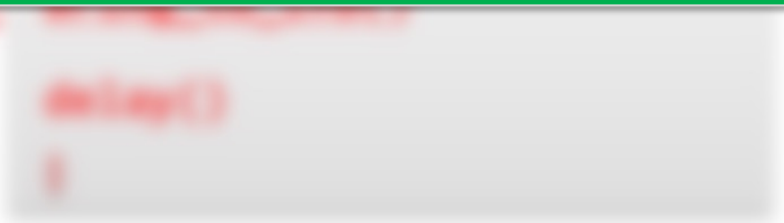
```
Bring_to_dcache()

Bring_to_lfb()

delay()
⋮
```

Helper & Setup gadgets

- These gadgets are **selected** by fuzzer based on the requirements of main gadgets







Execution Model

- **Execution Model** is the core of our **guided fuzzing** that helps with gadget selection to generate more useful inst. sequences
- **Predicts** the behavior of the fuzzed code by **estimating** the *micro-architectural* effects of adding **each** instruction to the output
- The execution of each inst. affects the ***state of the processor***
- The execution model **captures** these effects

Execution Model

- **Predicts** the behavior of the fuzzed code by **estimating** the micro-architectural effects of adding **each** instruction to the output

The execution model assists *gadget selection*, *instruction generation* and *leakage analysis*

Instruction Sequence Generation

Main Gadgets

Helper/Setup Gadgets

Randomly Choose

ASM Code

M3

Requirements:???

Step 1

Check EM

μ-arch is NOT ready!

Step 2

Carefully Choose

ASM Code

S2

H5

Step 3

EM Update

S2

H5

M3

Step 4

Fuzzer Output

ASM Code

S2

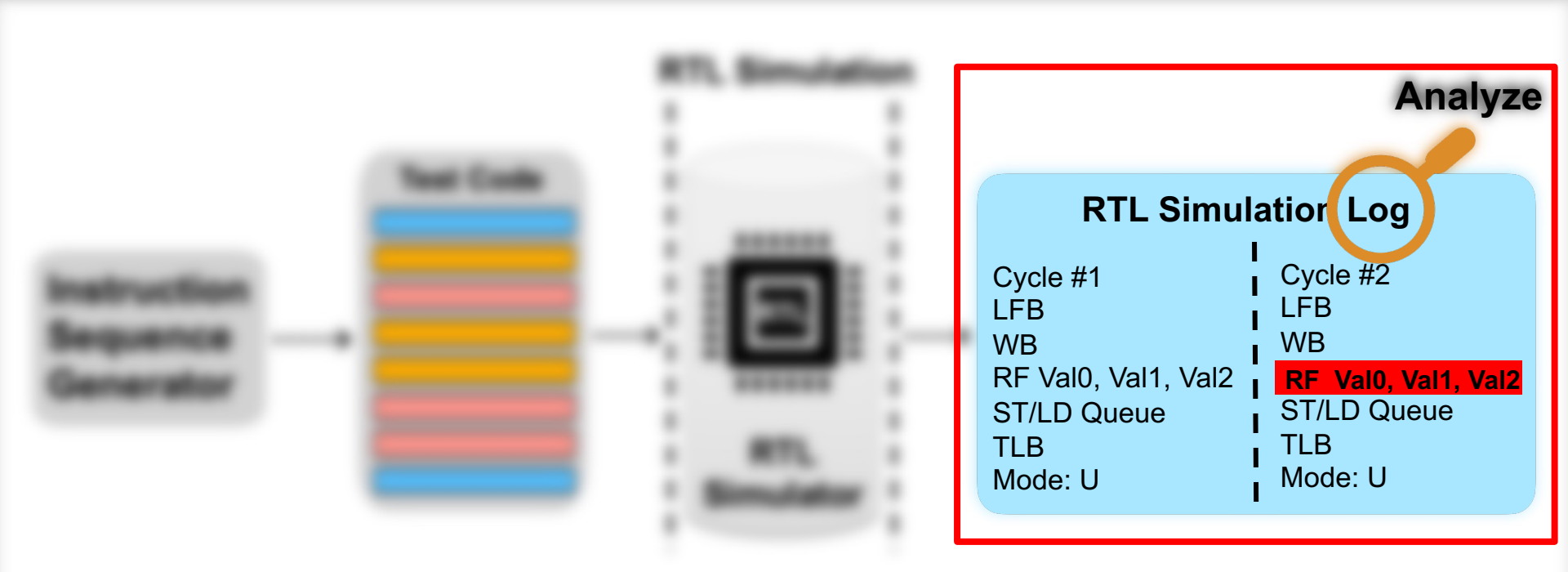
H5

M3

Outline

- IntroSpectre Framework
- Design
  - Instruction Sequence Generation
  - Leakage Analyzer
- Evaluation

IntroSpectre Framework



Leakage Analyzer

The Leakage Analyzer

RTL Simulation Log

Cycle #M	*Cycle #N*	*Cycle #P*	*Cycle #Q*	*Cycle #R*
LFB	LFB	LFB	LFB	LFB
WB	WB	WB	WB	WB
RF Val0, Val1, Val2	RF Val0, Val1, Val2	RF Val0, Val1, Val2	RF Val0, Val1, Val2	RF Val0, Val1, Val2
ST/LD Queue	ST/LD Queue	ST/LD Queue	ST/LD Queue	ST/LD Queue
TLB	TLB	TLB	TLB	TLB
Mode: U	Mode: U	Mode: U	Mode: U	Mode: U
•••	•••	•••	•••	•••

Here is the **simulation log** generated by the **RTL simulator**

How to look for secrets in the log?

The Leakage Analyzer

RTL Simulation Log

Cycle #M	*Cycle #N*	*Cycle #P*	*Cycle #Q*	*Cycle #R*
LFB	LFB	LFB	LFB	LFB
WB	WB	WB	WB	WB
RF Val0, Val1, Val2	RF Val0, Val1, Val2	RF Val0, Val1, Val2	RF Val0, Val1, Val2	RF Val0, Val1, Val2
ST/LD Queue	ST/LD Queue	ST/LD Queue	ST/LD Queue	ST/LD Queue
TLB	TLB	TLB	TLB	TLB
Mode: U	Mode: U	Mode: U	Mode: U	Mode: U
•••	•••	•••	•••	•••

A detailed **secret timeline** is **automatically** drawn using the ***execution model snapshots*** taken after adding **each** instruction to the fuzzer output

The Leakage Analyzer

RTL Simulation Log

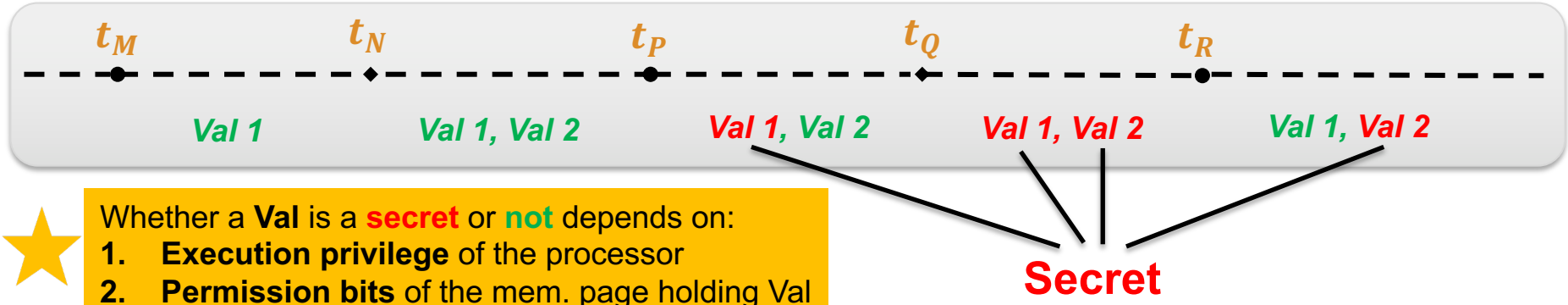
Cycle #M	*Cycle #N*	*Cycle #P*	*Cycle #Q*	*Cycle #R*
LFB	LFB	LFB	LFB	LFB
WB	WB	WB	WB	WB
RF Val0, Val1, Val2	RF Val0, Val1, Val2	RF Val0, Val1, Val2	RF Val0, Val1, Val2	RF Val0, Val1, Val2
ST/LD Queue	ST/LD Queue	ST/LD Queue	ST/LD Queue	ST/LD Queue
TLB	TLB	TLB	TLB	TLB
Mode: U	Mode: U	Mode: U	Mode: U	Mode: U
•••	•••	•••	•••	•••

This timeline ***specifies*** what values should be considered **secret** during different **stages** of program execution

The Leakage Analyzer

RTL Simulation Log

Cycle #M	*Cycle #N*	*Cycle #P*	*Cycle #Q*	*Cycle #R*
LFB	LFB	LFB	LFB	LFB
WB	WB	WB	WB	WB
RF Val0, Val1, Val2	RF Val0, Val1, Val2	RF Val0, Val1, Val2	RF Val0, Val1, Val2	RF Val0, Val1, Val2
ST/LD Queue	ST/LD Queue	ST/LD Queue	ST/LD Queue	ST/LD Queue
TLB	TLB	TLB	TLB	TLB
Mode: U	Mode: U	Mode: U	Mode: U	Mode: U
•••	•••	•••	•••	•••



Whether a **Val** is a **secret** or **not** depends on:
1. **Execution privilege** of the processor
2. **Permission bits** of the mem. page holding Val

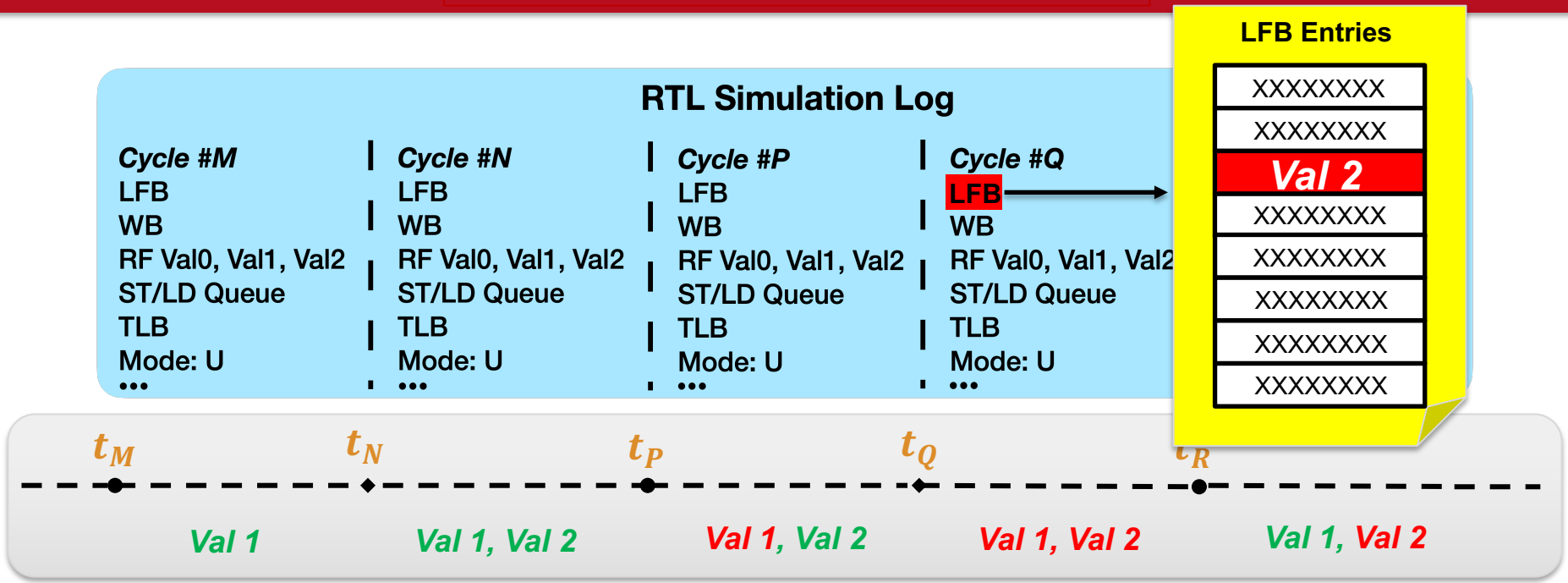
The Leakage Analyzer

RTL Simulation Log

Cycle #M	*Cycle #N*	*Cycle #P*	*Cycle #Q*	*Cycle #R*
LFB	LFB	LFB	**LFB**	LFB
WB	WB	WB	WB	WB
RF Val0, Val1, Val2	RF Val0, Val1, Val2	RF Val0, Val1, Val2	RF Val0, Val1, Val2	RF Val0, Val1, Val2
ST/LD Queue	ST/LD Queue	ST/LD Queue	ST/LD Queue	ST/LD Queue
TLB	TLB	TLB	TLB	TLB
Mode: U	Mode: U	Mode: U	Mode: U	Mode: U
•••	•••	•••	•••	•••

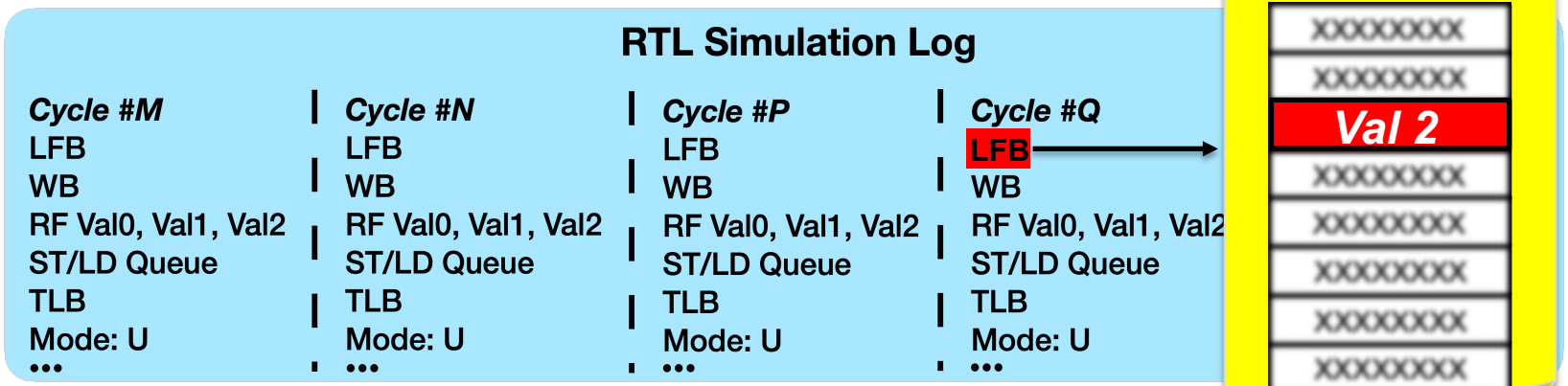
Let's assume that our analyzer finds ***Val 2*** in **LFB** during the execution cycles from t_Q to t_R.

The Leakage Analyzer



Let's assume that our analyzer finds ***Val 2*** in **LFB** during the execution cycles from t_Q to t_R.

The Leakage Analyzer



Secret Leakage Found!

Evaluation

Evaluation

- We evaluated **IntroSpectre** on **RISC-V BOOM v2.2.3** single core using **Verilator RTL Simulator v4.035**
- Out of around **100 fuzzing rounds**, discovered **13 distinct scenarios** leading to **secret leakage** through different **u-arch structures** categorized across three classes

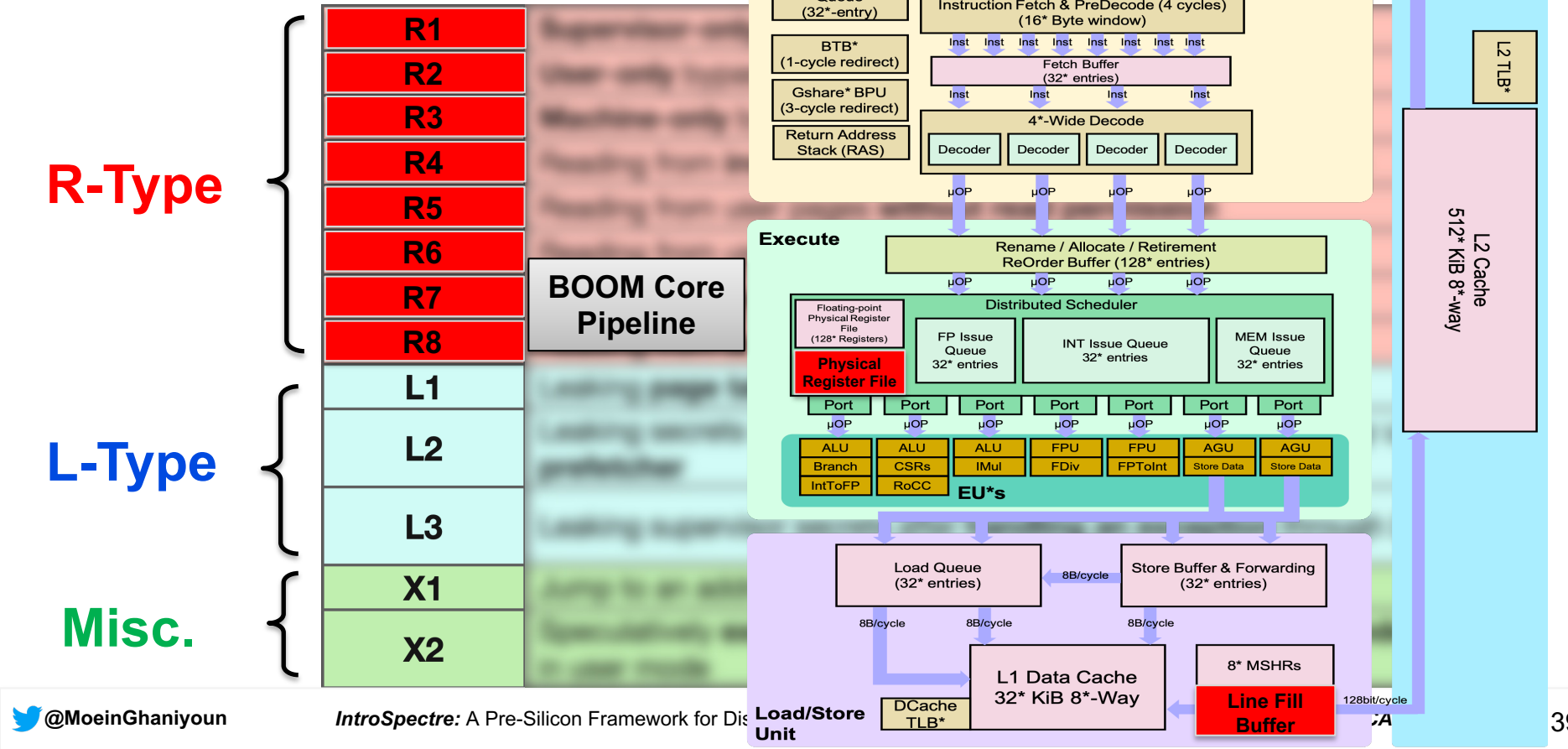
Many of these secret leakage scenarios have **NOT** been reported before on **BOOM**

- **R-Type Leakage:** Secret data in both **register file and LFB**
- **L-Type Leakage:** Secret data only in **LFB**
- **Miscellaneous:** Meltdown-JP & Meltdown-X

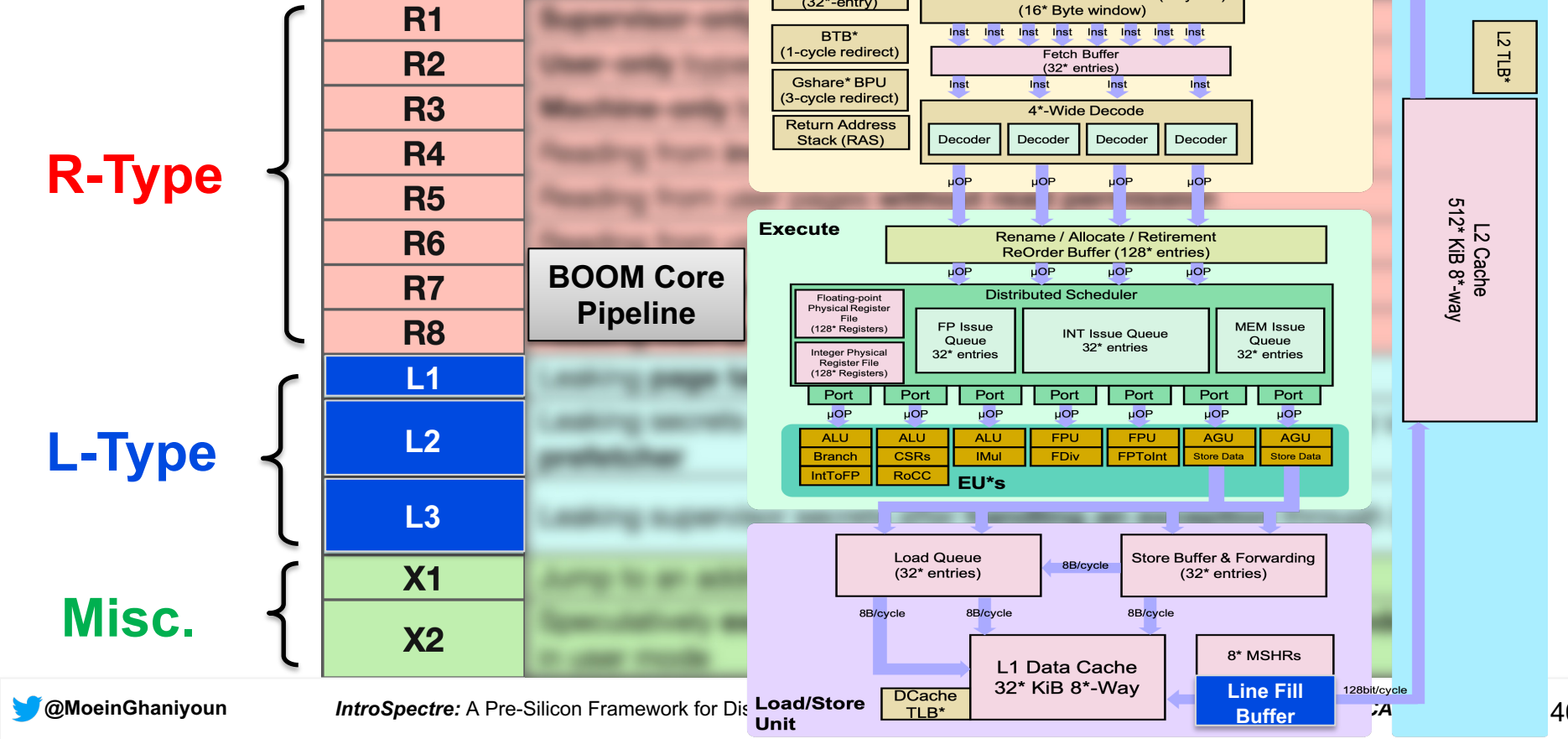
Leakage Scenarios in BOOM v2.2.3

R-Type	R1	**Supervisor-only** bypass
	R2	**User-only** bypass
	R3	**Machine-only** bypass
	R4	Reading from **invalid user pages** regardless of permission bits
	R5	Reading from user pages **without read permission**
	R6	Reading from user pages **with access and dirty bits off**
	R7	Reading from user pages **with access bit off**
	R8	Reading from user pages **with dirty bit off**
L-Type	L1	Leaking **page table entries** through LFB
	L2	Leaking secrets of a page without proper permissions in LFB by using **prefetcher**
	L3	Leaking supervisor secrets after **handling an exception** through LFB
Misc.	X1	Jump to an address and **execute the stale value**
	X2	Speculatively **execute supervisor-code/inaccessible-user-code** while in user mode

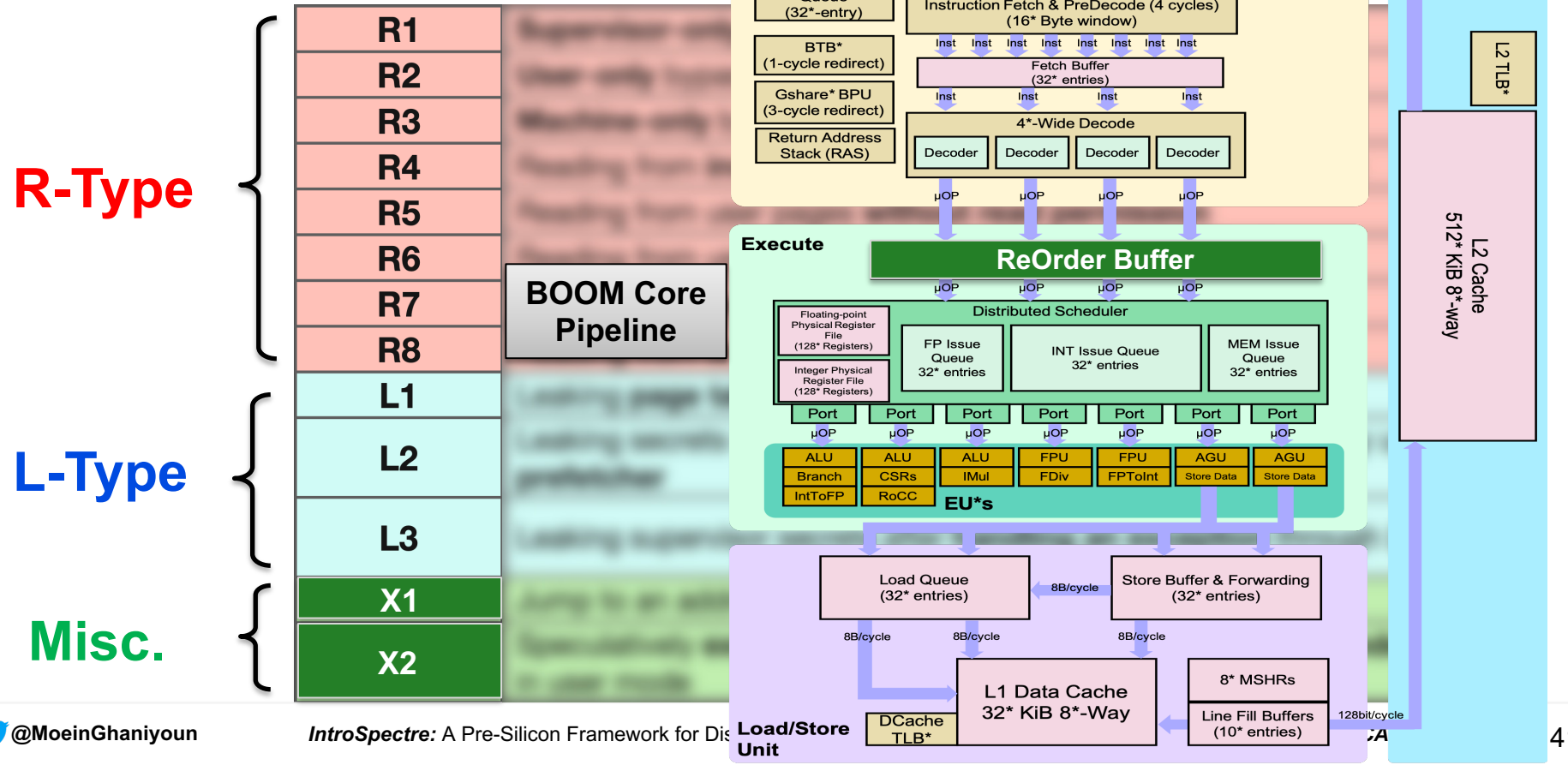
Leakage Scena



Leakage Scena



Leakage Scena



Leakage Scenarios in BOOM v2.2.3

R-Type	R1	**Supervisor-only** bypass
	R2	**User-only** bypass
	R3	**Machine-only** bypass
	R4	Reading from **invalid user pages** regardless of permission bits
	R5	Reading from user pages **without read permission**
	R6	Reading from user pages **with access and dirty bits off**
	R7	Reading from user pages **with access bit off**
	R8	Reading from user pages **with dirty bit off**
L-Type	L1	Leaking **page table entries** through LFB
	L2	Leaking secrets of a page without proper permissions in LFB by using **prefetcher**
	L3	Leaking supervisor secrets after **handling an exception** through LFB
Misc.	X1	Jump to an address and **execute the stale value**
	X2	Speculatively **execute supervisor-code/inaccessible-user-code** while in user mode

Leakage Scenarios in BOOM v2.2.3

R-Type	R1	**Supervisor-only** bypass
	R2	**User-only** bypass
	R3	**Machine-only** bypass
	R4	Reading from **invalid user pages** regardless of permission bits
	R5	Reading from user pages **without read permission**
	R6	Reading from user pages **with access and dirty bits off**
	R7	Reading from user pages **with access bit off**
	R8	Reading from user pages **with dirty bit off**
L-Type	L1	Leaking **page table entries** through LFB
	L2	Leaking secrets of a page without proper permissions in LFB by using **prefetcher**
	L3	Leaking supervisor secrets after **handling an exception** through LFB
Misc.	X1	Jump to an address and **execute the stale value**
	X2	Speculatively **execute supervisor-code/inaccessible-user-code** while in user mode

Bypassing all three privilege levels!

Leakage Scenarios in BOOM v2.2.3

R-Type	R1	**Supervisor-only** bypass
	R2	**User-only** bypass
	R3	**Machine-only** bypass
	R4	Reading from **invalid user pages** regardless of permission bits
	R5	Reading from user pages **without read permission**
	R6	Reading from user pages **with execute permission**
	R7	Reading from user pages **with access bit off**
	R8	Reading from user pages **with dirty bit off**
L-Type	L1	Leaking **page table entries** through LFB
	L2	Leaking secrets of a page without proper permissions in LFB by using **prefetcher**
	L3	Leaking supervisor secrets after **handling an exception** through LFB
Misc.	X1	Jump to an address and **execute the stale value**
	X2	Speculatively **execute supervisor-code/inaccessible-user-code** while in user mode

★ **Bypassing all three privilege levels!**

★ **Leaking secrets from inaccessible user pages!**

Leakage Scenarios in BOOM v2.2.3

R-Type	R1	**Supervisor-only** bypass
	R2	**User-only** bypass ★ **Bypassing all three privilege levels!**
	R3	**Machine-only** bypass
	R4	Reading from **invalid user pages** regardless of permission bits
	R5	Reading from user pages **without read permission**
	R6	Reading ... ★ **Leaking secrets from inaccessible user pages!**
	R7	Reading from user pages **with access bit of**
	R8	Reading from user pages **with dirty bit off**
L-Type	L1	Leaking **page table entries** through LFB
	L2	Leaking secrets of a page without proper permissions in LFB by using **prefetcher** ★ **Potential secret leakage from LFB!**
	L3	Leaking supervisor secrets after **handling an exception** through LFB
Misc.	X1	Jump to an address and **execute the stale value**
	X2	Speculatively **execute supervisor-code/inaccessible-user-code** while in user mode

Leakage Scenarios in BOOM v2.2.3

Coverage Analysis

- The coverage of **IntroSpectre** can be examined along four dimensions:

1. **Coverage of u-arch structures**

   ★ All relevant **u-arch structure' state** is recorded in the simulation log

2. **Coverage of isolation boundaries**

   ★ All possible accesses across **all isolation boundaries** are covered

3. **Coverage of gadgets**

   ★ All **known and applicable** Meltdown-like attacks are covered + other **torturing** gadgets

4. **False positives/ False negatives**

   ★ **No** false negatives regarding leakage scenarios triggered by fuzzer

   ★ **Cannot** guarantee to capture the leakage scenarios not exercised by the test code

   ★ **Might** have false positives regarding the exploitability of a secret leakage

Summary

- We developed **IntroSpectre**, a framework for **transient execution vulnerability detection**
- **Full visibility** into internal processor's state enables a **systematic evaluation** of the design
- We focus on **Meltdown-like** leaks, but the tool can be **extended** to cover **other** transient execution attacks (**Spectre**)

Thank you!

★The source code of **IntroSpectre** will be available soon at :★

GitHub https://github.com/MoeinGhaniyoun/IntroSpectre

@MoeinGhaniyoun Ghaniyoun.1@osu.edu

COMPUTER ARCHITECTURE RESEARCH LAB

http://arch.cse.ohio-state.edu