Restricting Control Flow During Speculative Execution with Venkman

Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell

Department of Computer Science
University of Rochester
Speculative Execution

• **Feature to improve processor performance**
  - Equipped in x86, ARM, POWER, etc

• **Execute instructions prior to knowing if they are needed**
  - Restore and re-execute on mis-speculation
  - Leave observable side effects

• **Vulnerable to sophisticated attacks!**
Spectre Attacks

- **Variant 1: exploit conditional branches**
  - Direction prediction using PHT
  - Mitigations: fence, data dependence

- **Variant 2: exploit indirect branches**
  - Target prediction using BTB and RSB
  - Mitigations: microcode, Retpoline

```c
if (x < arr1_size) {
    y = arr1[x];
}

(*func_ptr)(); // return;

if (x < arr1_size) {
    load_fence();
    y = arr1[x];
}
```
What if direct branch targets can also be poisoned?

```assembly
malicious_target:
    call 2
    pause
    jmp 1
    mov %r11, (%rsp)
    ret
    ...
```

Retpoline becomes vulnerable too!
Venkman: Our Solution!

- **Defense against BTB & RSB poisoning**
  - Aligned control-flow transfer targets
  - Protective instructions not bypassed

- **Broad threat model**
  - Any program can be a potential attacker!
  - Require whole-system instrumentation
Outline

• Design
• Implementation
• Security Evaluation
• Space & Performance Evaluation
• Conclusions & Future Work
Venkman System Architecture

LLVM Code for Potential Victim → Native Code Generation → Protection Transform → Alignment Transform → Alignment Verifier → TCB

LLVM Code for Unprotected Program → Native Code Generation → Alignment Transform

Binary Code

Execute Binary Code

Pass

Fail

Deny Binary Execution
Venkman Transformations

• **Alignment transformations**
  - Code padding & alignment
  - Bit-masking control data

• **Protection transformations**
  - Spectre protection

} Core of Venkman
Code Padding & Alignment

- **Transform basic blocks into bundles**
  - Groups of Instructions sized and aligned at $2^s$ bytes
  - Split large BBs
  - Pad NOPs to small BBs
- **Function calls at the end of bundle**
Honor co-location requirements

Bundle 1

... 
Protected instruction(s)
Bundle 2
Protected instruction
...
Bit-Masking Control Data

- **Clear lower $S$ bits**
- **Clear higher $(64 - T)$ bits**

\[ S = 4, \quad T = 32 \]

- **Transform branches w/ in-memory target**

  ```assembly
  call *%reg
  and $0x00000000ffffffff0, %reg
  call *%reg
  ```

  ```assembly
  ret
  S = 4, \quad T = 32
  ```

  ```assembly
  pop %reg
  and $0x00000000ffffffff0, %reg
  jmp *%reg
  ```
• **Insert Spectre-resistant SFI** [1]

![Diagram showing the insertion of Spectre-resistant SFI](image)

Spectre Protection

- **Insert fences**

![Diagram showing Spectre Protection with fences]

- First load
- Second load

- First load
- Second load
- Fence
Outline

- Design
- Implementation
- Security Evaluation
- Space & Performance Evaluation
- Conclusions & Future Work
Implementation

- Implemented on POWER architecture
- Extended LLVM with MachineFunctionPasses
- 32-byte bundles \( (S = 5) \)
- Code segment at first 32 TB \( (T = 45) \)
- Use EIEIO as fence
- Use dummy SFI
Outline

- Design
- Implementation
- **Security Evaluation**
- Space & Performance Evaluation
- Conclusions & Future Work
Security Evaluation

- **Proof-of-concept Spectre V2 attack on POWER**

Attacker:

- For all secret bytes:
  - For N iterations:
    - Train BTB
    - Victim execution
    - Probe secret byte value via cache side channel
    - Record most likely byte values

Victim:

- Perform an indirect branch to malicious code address chosen by attacker
- Access secret byte speculatively

With Venkman, such attack no longer works!
Outline

● Design
● Implementation
● Security Evaluation
● **Space & Performance Evaluation**
● Conclusions & Future Work
Experimental Setup

- **Hardware specifications**
  - 64-bit IBM POWER8
  - 20 cores, 8 threads/core
  - 4.1 GHz
  - 64 GB RAM

- **Software specifications**
  - CentOS 7 w/ Linux 3.10.0
  - LLVM/Clang 4.0.1

- **Configurations**
  - Baseline
  - Alignment only
  - Alignment + CFI (Venkman)
  - Venkman + Fence
  - Venkman + SFI-Load

- **Benchmarks & applications**
  - SPEC CPU 2017
  - Nginx 1.15.8
  - GnuPG 1.4.23
  - ClamAV 0.92
Code Size Overhead on SPEC CPU 2017
Nginx File Transfer Rate

Baseline

Venkman Normalized to Baseline
GnuPG & ClamAV Overhead

GnuPG Encryption

ClamAV
Outline

- Design
- Implementation
- Security Evaluation
- Space & Performance Evaluation
- Conclusions & Future Work
Conclusions

- Venkman mitigates BTB & RSB poisoning
- When protecting fences, Venkman exhibits
  - 1.64x average space overhead
  - 2.97x average performance overhead
- When protecting SFI, Venkman exhibits
  - 2.13x average space overhead
  - 1.43x average performance overhead
- 61.3% space and 8.6% performance overhead for non-victim programs
Future Work

- Reduce overhead
- Port to x86 & ARM
Thank you!