Security-Aware Processor Architecture Design
CS 6501 – Fall 2018
Ashish Venkat
Agenda

• Theme Selection (due today at 11:59:59pm)
• Readings and Presentation Logistics
• Quick Processor Architecture Review (continued from Tuesday)
• Research Themes
Readings and Presentation Logistics

• If your group is presenting a topic, make sure a group member posts the papers you’d like us to read on Piazza.
• Assign 2 papers per lecture.
• Give us at least two full days to read the papers – if you’re presenting on Tuesday, post the papers on Saturday at 11:59:59pm.
• If this is not a paper on the project descriptions document, consult with me before posting.
• Presentation guidelines:
  • Don’t spend too much time describing the paper.
  • Critique the paper: 5 positive aspects, 5 negative aspects, 5 questions you have about the paper.
  • If it is an attack paper, talk about potential mitigations. If it is a defense paper, talk about potential backdoor attacks.
  • Ask questions to the audience and spark discussion.
Recap from last class

• 5-stage MIPS Pipeline
• Superscalar execution
• RISC vs CISC
• Microcode
• Branch Prediction and OOO Execution
• Ill-effects of Speculative Execution??
Buffer Overflow Exploits – Code Injection

Malicious Code
xor %eax, %eax
mov $0x1, %al
xor %ebx, %ebx
int $0x80

Application Code
lea -0x78(%ebp),%eax
mov %eax,0x8(%esp)
mov -0x80(%ebp),%eax
mov %eax,0x4(%ebp)
movl $0x3,(%esp)
jmp *%eax

Inject malicious code on stack/heap and subvert control flow
Buffer Overflow Exploits – Code Reuse

Bad Behavior

Malicious Code

PC

xor %eax, %eax
mov $0x1, %al
xor %ebx, %ebx
int $0x80

Application Code

lea -0x78(%ebp),%eax
mov %eax,0x8(%esp)
mov -0x80(%ebp),%eax
mov %eax,0x4(%ebp)
movl $0x3(%esp)
jmp *%eax

Return-Oriented Programming
Return-Oriented Programming

Read only Text Section

lea -0x78(%ebp),%eax
mov %eax,0x8(%esp)
call d92e0 <memcpy>
....
ret
....

mov %edx,-0x94(%ebp)
movl $0x3,(%esp)
mov %eax,0x4(%esp)
ret
....
....
xor %eax,%eax
ret
....
....
pop %ebx
ret

Stack

0x20d1b0
0x17049d
0x10ad

Register State

<table>
<thead>
<tr>
<th>eax</th>
<th>ebx</th>
<th>ecx</th>
<th>edx</th>
<th>esp</th>
<th>ebp</th>
<th>esi</th>
<th>edi</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
</tr>
</tbody>
</table>

Dynamic Execution Stream

lea -0x78(%ebp),%eax
mov %eax,0x8(%esp)

Caller Frame ends here
Return-Oriented Programming

Read only Text Section

lea  -0x78(%ebp),%eax
mov   %eax,0x8(%esp)
call d92e0 <memcpy>
 ....
ret
 ....

mov   %edx,-0x94(%ebp)
movl  $0x3,(%esp)
mov   %eax,0x4(%esp)
ret
 ....
 ....
xor   %eax,%eax
ret
 ....
 ....

pop   %ebx
ret

Exploit buffer overflow

Stack

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x870f65</td>
</tr>
<tr>
<td>0x87098d</td>
</tr>
<tr>
<td>0xbc</td>
</tr>
<tr>
<td>0x870234</td>
</tr>
<tr>
<td>0x432a123</td>
</tr>
<tr>
<td>0x65708ad6</td>
</tr>
</tbody>
</table>

Register State

<table>
<thead>
<tr>
<th>eax</th>
<th>ebx</th>
<th>ecx</th>
<th>edx</th>
<th>esp</th>
<th>ebp</th>
<th>esi</th>
<th>edi</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
</tr>
</tbody>
</table>

Dynamic Execution Stream

lea  -0x78(%ebp),%eax
mov   %eax,0x8(%esp)
call d92e0 <memcpy>
 ....

Caller Frame ends here

Gadgets

 eax ebx ecx edx esp ebp esi edi

XXX XXX XXX XXX XXX XXX XXX XXX

XXX XXX XXX XXX XXX XXX XXX XXX
Return-Oriented Programming

Read only Text Section

le a -0x78(%ebp),%eax
mov %eax,0x8(%esp)
call d92e0 <memcpy>
....
ret
....

mov %edx,-0x94(%ebp)
movl $0x3,(%esp)
mov %eax,0x4(%esp)
ret
....
....
xor %eax,%eax
ret
....
....
pop %ebx
ret

Return to Gadget 1

Stack

<table>
<thead>
<tr>
<th>Stack Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x870f65</td>
</tr>
<tr>
<td>0x87098d</td>
</tr>
<tr>
<td>0xbc0d</td>
</tr>
<tr>
<td>0x870234</td>
</tr>
<tr>
<td>0x432a123</td>
</tr>
<tr>
<td>0x65708ad6</td>
</tr>
</tbody>
</table>

Register State

<table>
<thead>
<tr>
<th>eax</th>
<th>ebx</th>
<th>ecx</th>
<th>edx</th>
<th>esp</th>
<th>ebp</th>
<th>esi</th>
<th>edi</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXX</td>
<td>bcd</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
</tr>
</tbody>
</table>

Dynamic Execution Stream

lea -0x78(%ebp),%eax
mov %eax,0x8(%esp)
call d92e0 <memcpy>
....
ret
pop %ebx
Return-Oriented Programming

Read only Text Section

Gadgets

PC

Return to Gadget 2

Stack

Dynamic Execution Stream

Register State

<table>
<thead>
<tr>
<th>eax</th>
<th>ebx</th>
<th>ecx</th>
<th>edx</th>
<th>esp</th>
<th>ebp</th>
<th>esi</th>
<th>edi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bcd</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
</tr>
</tbody>
</table>

lea -0x78(%ebp),%eax
mov %eax,0x8(%esp)
call d92e0 <memcpy>
....
ret
....
lea -0x94(%ebp),%edx
mov $0x3,(%esp)
mov %eax,0x4(%esp)
ret
....
....
xor %eax,%eax
ret
....
pop %ebx
ret

lea -0x78(%ebp),%eax
mov %eax,0x8(%esp)
call d92e0 <memcpy>
....
ret
pop %ebx
xor %eax,%eax
Return-Oriented Programming

Read only Text Section

```
lea -0x78(%ebp),%eax
mov %eax,0x8(%esp)
call d92e0 <memcpy>
....
ret
....
mov %edx,-0x94(%ebp)
movl $0x3,(%esp)
mov %eax,0x4(%esp)
ret
....
....
xor %eax,%eax
ret
....
....
pop %ebx
ret
```

Return to Gadget 3

Stack

```
lea -0x78(%ebp),%eax
mov %eax,0x8(%esp)
call d92e0 <memcpy>
....
ret
....
mov %edx,-0x94(%ebp)
movl $0x3,(%esp)
mov %eax,0x4(%esp)
ret
```

Register State

<table>
<thead>
<tr>
<th>eax</th>
<th>ebx</th>
<th>ecx</th>
<th>edx</th>
<th>esp</th>
<th>ebp</th>
<th>esi</th>
<th>edi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bcd</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
</tr>
</tbody>
</table>

Dynamic Execution Stream

```
lea -0x78(%ebp),%eax
mov %eax,0x8(%esp)
call d92e0 <memcpy>
....
ret
pop %ebx
xor %eax,%eax
mov %edx,-0x94(%ebp)
movl $0x3,(%esp)
mov %eax,0x4(%esp)
```
Escape from ROP

ROP thrives on 3 fundamental foundations:

• Buffer overflow vulnerabilities
• Ability to hijack control flow
• Prior knowledge of gadget locations
Security Implications of Speculative Execution

if \( x < \text{array1\_size} \)
\[
y = \text{array2}[\text{array1}[x] \times 256];
\]

• Upon branch misspeculation, the bounds check is bypassed and the Spectre gadget executes.
• The spectre gadget leaks information by establishing an observable cache footprint.
• Impacts nearly every computer in the world.
Security Implications of Speculative Execution

• What are the potential ill effects of speculative execution?
  • Spectre-v1: Bypass bounds check and leak sensitive information along mis-speculated path.
  • Meltdown: Execute privileged code along mis-speculated path while lacking sufficient privileges.

• What happens if an attacker controls branch prediction?
  • Spectre-v2: Speculatively jump to and execute arbitrary attacker-intended code.
  • Spectre-v5: Mispredict function call return and enable speculative chaining of malicious ROP-style gadgets.
  • Branchscope: Leak secret keys by inferring branch direction.
# Spectre Variants

<table>
<thead>
<tr>
<th>Variant</th>
<th>CVE</th>
<th>Vulnerability Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spectre v1</td>
<td>2017-5753</td>
<td>Bounds Check Bypass (BCB)</td>
</tr>
<tr>
<td>Spectre v2</td>
<td>2017-5715</td>
<td>Branch Target Injection (BTI)</td>
</tr>
<tr>
<td>Spectre v3</td>
<td>2017-5754</td>
<td>Rogue Data Cache Load (RDCL)</td>
</tr>
<tr>
<td>Spectre v3a</td>
<td>2017-3640</td>
<td>Rogue System Register Read (RSRD)</td>
</tr>
<tr>
<td>Spectre v4</td>
<td>2017-3639</td>
<td>Speculative Store Bypass (SSB)</td>
</tr>
<tr>
<td>Spectre-NG v3</td>
<td>2017-3665</td>
<td>Lazy FP State Restore</td>
</tr>
<tr>
<td>Spectre v1.1</td>
<td>2018-3693</td>
<td>Bounds Check Bypass Store (BCBS)</td>
</tr>
<tr>
<td>Spectre v1.2</td>
<td>-</td>
<td>Read-only Protection Bypass</td>
</tr>
<tr>
<td>Spectre v5</td>
<td>-</td>
<td>ret2spec and SpecRSB</td>
</tr>
<tr>
<td>NetSpectre</td>
<td>-</td>
<td>Remote Bounds Check Bypass</td>
</tr>
</tbody>
</table>
Simultaneous Multithreading (SMT)

• Different threads make different amount of progress through execution
• Simultaneously execute instructions from different software threads to improve CPU utilization
• Multiple hardware contexts for each software thread.
• Share other resources – fetch/issue slots, queues, FUs, etc.
Security implications of SMT

• What happens if a spy thread is co-located with a victim thread?
• Can it secretly infer the execution characteristics of the victim thread?
• What are the sources of information leak?
• Can it influence the branch outcomes of the victim thread?
Cache side-channel attacks
Cache side-channel attacks
Cache side-channel attacks

- Pre-Attack
- Process
- Shared Data Cache
- Sensitive Computation (Key-Dependent Data Access)
Cache side-channel attacks

Pre-Attack

Processor

Shared Data Cache

Sensitive Computation
(Key-Dependent Data Access)

Leaves Memory Signature

Attacker Process

Victim Process
Cache side-channel attacks

Processor

Attacker Process

Pre-Attack

Shared Data Cache

Processor

Victim Process

Sensitive Computation (Key-Dependent Data Access)

Probing Cache Lines

Leaves Memory Signature
Hardware Specialization

- Plethora of performance accelerators
- Execution latencies vary
- Cache organizations differ
- A number of potential side channels
- Security implications understudied
Agenda

• Theme Selection (due today at 11:59:59pm)
• Readings and Presentation Logistics
• Quick Processor Architecture Review (continued from Tuesday)
• Research Themes
Research Themes Sneak Peek

• **Theme 1: Branch Predictor Hardening**
  - Spectre attacks rely on mistraining the branch predictor to speculatively execute attacker-intended code and leak information via side channels.
  - High-risk and high-impact.
  - Branchscope attacks can further leak branch direction information and subvert SGX protection.
  - Modern front-ends are pretty deep – you really take a while to recover.
  - We need branch predictors that can be efficiently trained, but are resilient against mistraining.
  - A number of approaches possible: partitioning, fault isolation, randomization, adversarial/secure machine learning, etc.
Research Themes Sneak Peek

• **Theme 2: GPU Memory Attack**
  
  • GPUs are ubiquitous and yet GPU security research has been fairly scant.
  
  • Current attacks steal information from uninitialized memory.
  
  • Research Questions:
    
    • Can a CPU thread steal secrets from GPU memory?
    
    • What side channels are available to the attacker?
    
    • Is the attack easy to mount?

  • HSA’s unified address space could exacerbate this problem and potentially lead to new high-resolution attacks.
Research Themes Sneak Peek

• **Theme 3: SMT Contention Characterization**
  
  • Simultaneous Multithreading (SMT) can substantially improve CPU utilization.
  
  • However, SMT is vulnerable to a suite of microarchitectural side channel attacks. In fact, Intel advises disabling SMT if you care about security.
  
  • Can we isolate microarchitectural structures that could potentially leak information?
  
  • Can we propose mechanisms to characterize the contention for microarchitectural structures? (e.g., a perceptron that learns from patterns)
  
  • Can we detect information leakage by observing certain patterns? Can this be learned/unlearned?
Research Themes Sneak Peek

• **Theme 4: Formal Verification of Microcode Updates**

  • Most modern processors including Intel, AMD, and ARM implement translated instruction sets (CISC $\rightarrow$ microcode RISC).
  
  • Intel and AMD further allow for field updates for fixing errata.
  
  • Several microcode updates (MCU) have been announced to mitigate Spectre. The first such MCU failed and was quickly retracted by Microsoft.
  
  • More research calls for a flexible microcode customization scheme (ISCA 2018, CCS 2018) via microcode updates that expose API – recipe for microcode injection.
  
  • Can we formally verify microcode updates to allow for flexible, yet secure microcode customization?
Research Themes Sneak Peek

• **Theme 5: Context-Sensitive Capability Protection**
  
  • Fat pointers contain bounds and permissions info in addition to addresses.
  
  • Fat pointer dereferences are validated by special capability loads and stores that check for invalid accesses.
  
  • Capabilities are easy to implement on a RISC ISA since all ALU operations happen within registers.
  
  • How does this translate to the x86 micro-op ISA?
  
  • Can we make this context-sensitive – the same CISC instruction gets translated into two different versions, one with capabilities and another without, depending upon whether we’re executing sensitive code?
Research Themes Sneak Peek

• Theme 6, 7, 8 ...
  • You are free to suggest and explore more project topics/themes as long as you convince me of their novelty and relevance.
Research Themes Sneak Peek

• These themes are open-ended for you to be creative.
• Different student groups may choose to work on the same theme. However, their approaches are expected to differ.
• In research, you might get scooped – always important to put out your (well-constructed) idea/architecture sooner than later.
• Sometimes, your idea may not pan out – keep looking for different ways to spin your idea.
• Good papers eventually get published and noticed.
Security-Aware Processor Architecture Design
CS 6501 – Fall 2018
Ashish Venkat