

Exploiting Branch Target Injection

Jann Horn, Google Project Zero

Outline

- Introduction
- Reverse-engineering branch prediction
- Leaking host memory from KVM

Disclaimer

- I haven't worked in CPU design
- I don't really understand how CPUs work
- Large parts of this talk are based on guesses
- This isn't necessarily how all CPUs work

Variants overview

Spectre

- CVE-2017-5753
 - Variant 1
 - Bounds Check Bypass
 - Primarily affects interpreters/JITs
- CVE-2017-5715
 - Variant 2
 - Branch Target Injection
 - Primarily affects kernels/hypervisors

Meltdown

- CVE-2017-5754
- Variant 3
- Rogue Data Cache Load
- Affects kernels (and architecturally equivalent software)

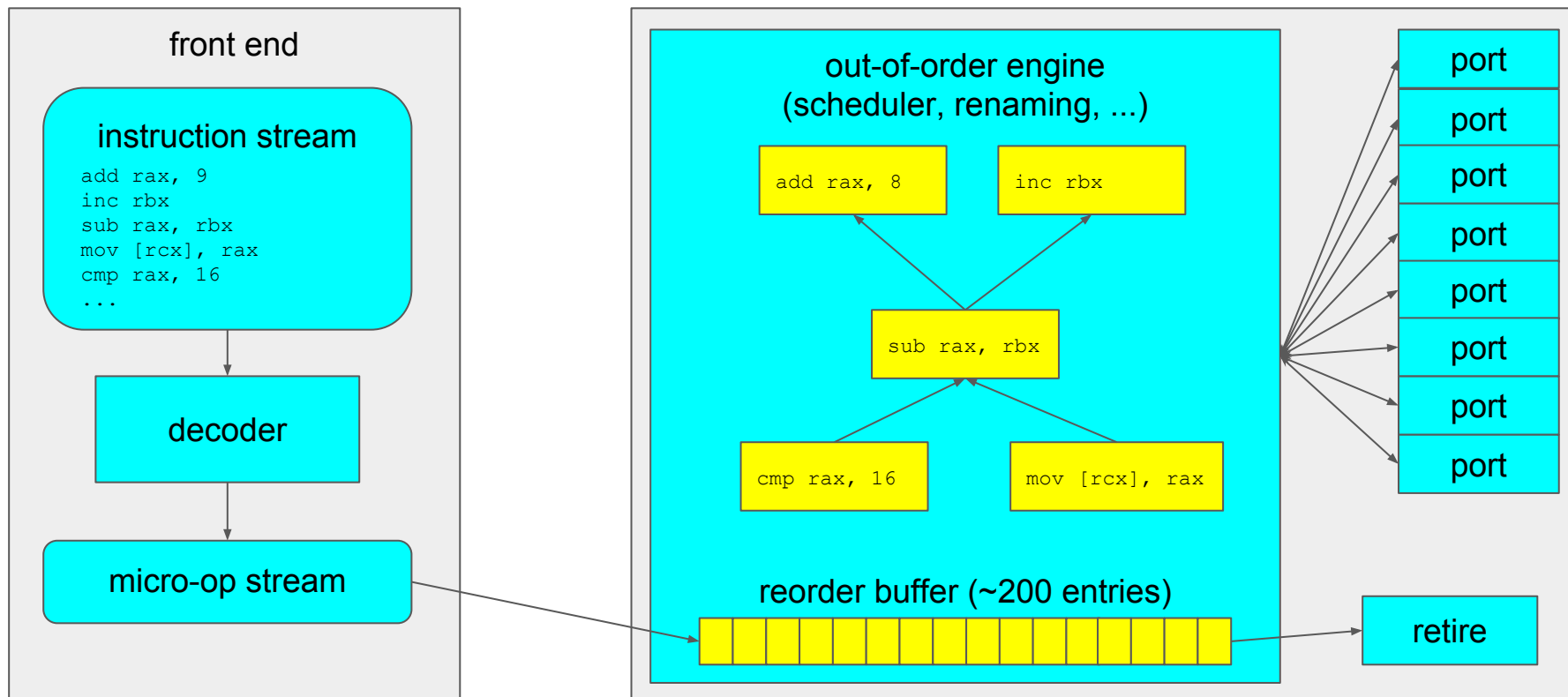
Performance

- Modern consumer CPU clock rates: ~4GHz
 - Memory is slow: ~170 clock cycles latency on my machine
 - CPU needs to work around high memory access latencies
 - Adding parallelism is easier than making processing faster
 - CPU needs to do things in parallel for performance
-
- Performance optimizations can lead to security issues!

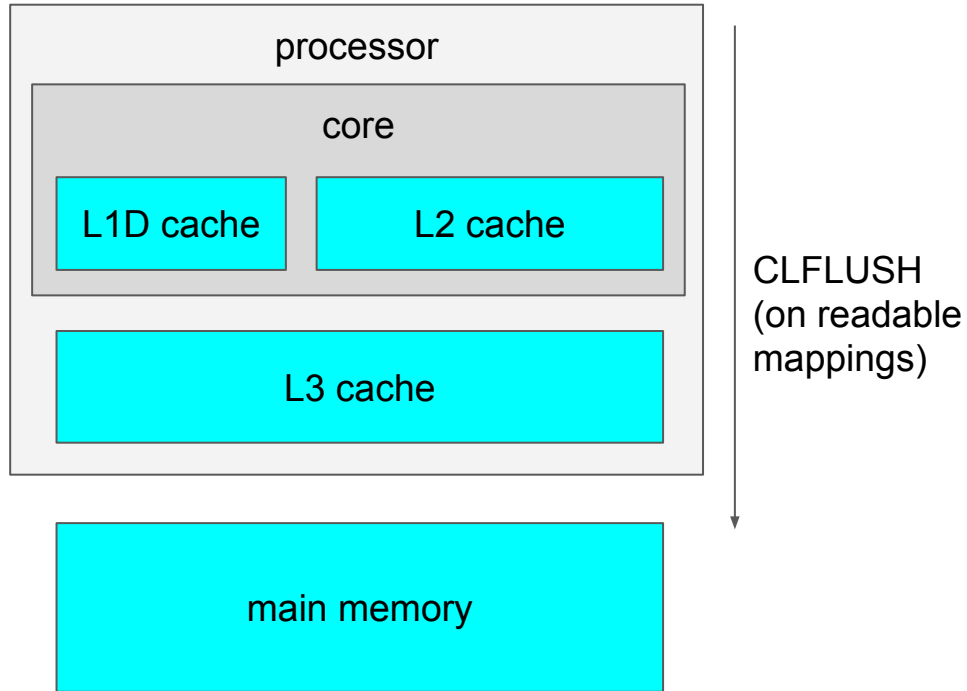
Performance Optimization Resources

- everyone wants programs to run fast
- processor vendors want application authors to be able to write fast code
- architectural behavior requires architecture documentation; *performance optimization requires microarchitecture documentation*
- if you want information about microarchitecture, read performance optimization guides
- Intel: <https://software.intel.com/en-us/articles/intel-sdm#optimization> ("optimization reference manual")
- AMD: <https://developer.amd.com/resources/developer-guides-manuals/> ("Software Optimization Guide")

Out-of-order execution



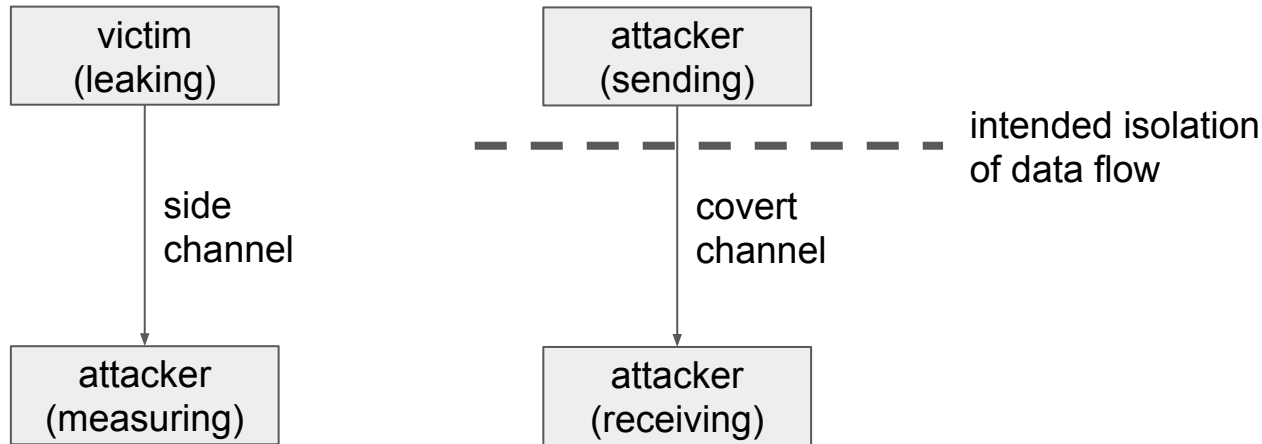
Data caching



- caches store memory in chunks of 64 bytes ("cache lines")
- multiple levels of cache
- L1D is fast, L3 is slower, main memory is very slow

Side Channels, Covert Channels

- performance/timing of process A is affected by process B
- side channel: process A can infer what process B is doing (uncooperatively)
- covert channel: process B can deliberately transmit information to process A
- side channels can often also be used as covert channels

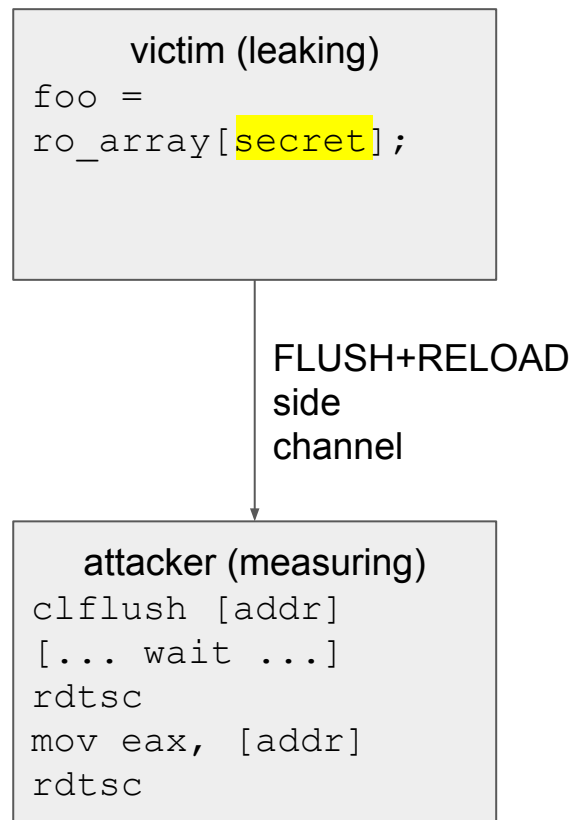


Side Channels, Covert Channels: FLUSH+RELOAD

For measuring accesses to shared read-only memory (.rodata / .text / zero page / vsyscall page / ...):

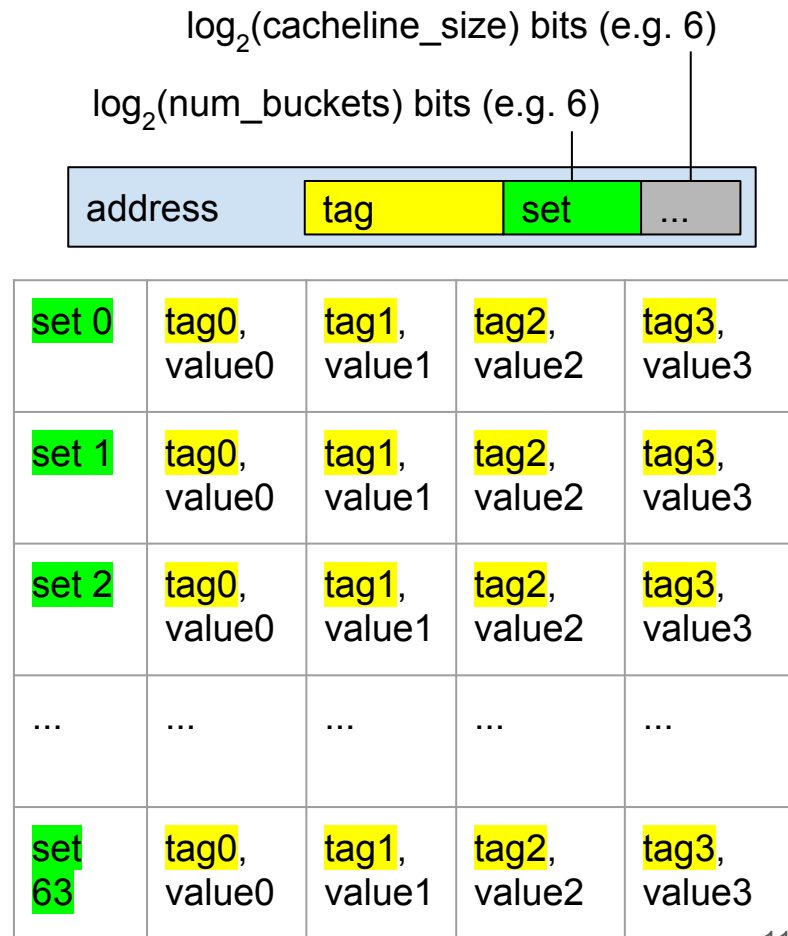
1. process A flushes cache line using CLFLUSH
2. process B maybe accesses cache line
3. process A accesses cache line, measuring access time

Limited applicability, but simple and fast



N-way caches; Eviction

- used in data caches and elsewhere
- software equivalent: think "hashmap with fixed-size arrays as buckets"
- fixed size: adding new entries removes older ones
- attacker can flush a set from the cache by adding new entries (*eviction strategy*)
 - strategy for Intel L3 caches described in the [rowhammer.js](#) paper by Daniel Gruss, Clémentine Maurice, Stefan Mangard
- (simplified: Intel L3 set selection is more complex, see research by Clementine Maurice et al.)



Branch Prediction

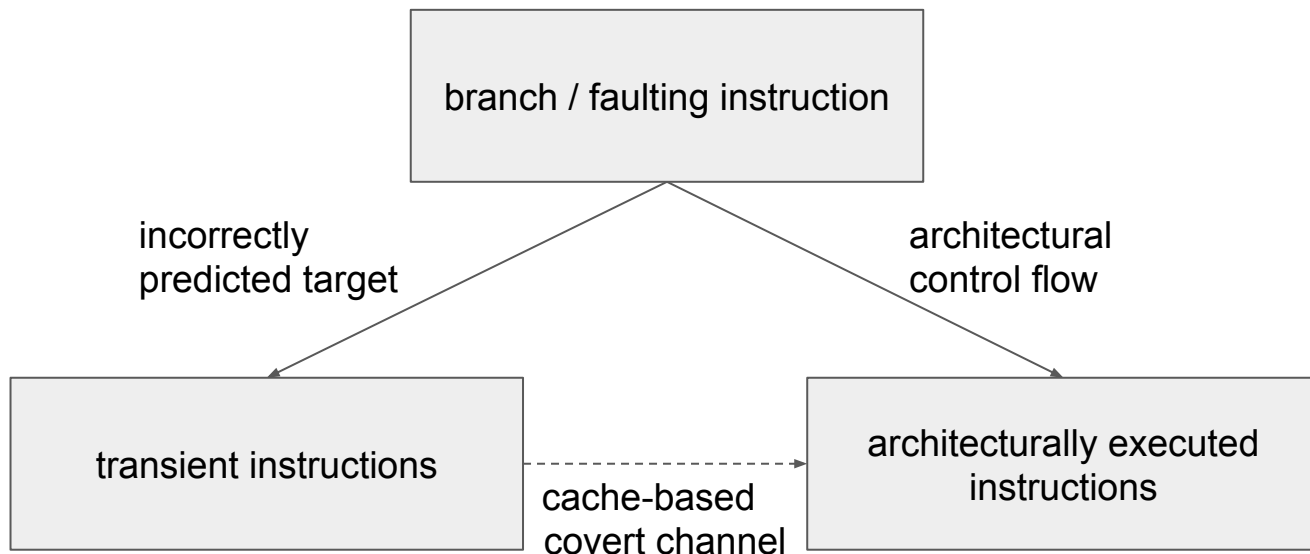
- processor predicts outcomes of branches
- predictions are based on previous behavior
- predictions help with executing more things in parallel

Misspeculation

- Exceptions and incorrect branch prediction can cause “rollback” of *transient instructions*
- Old register states are preserved, can be restored
- Memory writes are buffered, can be discarded
- Intuition: Transient instructions are sandboxed
- Covert channels matter
- **Cache modifications are not restored!**

Covert channel out of misspeculation

- Sending via FLUSH+RELOAD covert channel works from transient instructions



Variant 1: Abusing conditional branch misprediction

```
struct array {
    unsigned long length;
    unsigned char data[];
};
struct array *arr1 = ...; /* array of size 0x100 */
struct array *arr2 = ...; /* array of size 0x400 */
/* >0x100 (OUT OF BOUNDS!) */
unsigned long untrusted_index = ...;
if (untrusted_index < arr1->length) {
    char value = arr1->data[untrusted_index];
    unsigned long index2 = ((value&1)*0x100)+0x200;
    unsigned char value2 = arr2->data[index2];
}
```

mispredicted branch;

->length read must be slow!

speculatively unbounded read

sending on covert channel

Branch Prediction: Other patterns (UNTESTED)

- type check
- NULL pointer dereference
- out-of-bounds access into object table with function pointers

```
struct foo_ops {
    void (*bar)(void);
};

struct foo {
    struct foo_ops *ops;
};

struct foo **foo_array;
size_t foo_array_len;

void do_bar(size_t idx) {
    if (idx >= foo_array_len) return;
    foo_array[idx]->ops->bar();
}
```


Indirect Branches

- instruction stream does not contain target addresses
- target must be fetched from memory
- CPU will speculate about branch target

```
kvm_x86_ops->handle_external_intr(vcpu);
```

```
struct kvm_x86_ops *kvm_x86_ops;
```

```
static struct kvm_x86_ops vmx_x86_ops = {  
    [...]  
    .handle_external_intr =  
    vmx_handle_external_intr,  
    [...]  
};
```

[code simplified]

Variant 2: Basics

- Branch predictor state is stored in a Branch Target Buffer (BTB)
 - Indexed and tagged by (on Intel Haswell):
 - partial virtual address
 - recent branch history fingerprint [sometimes]
- Branch prediction is expected to sometimes be wrong
- Unique tagging in the BTB is unnecessary for correctness
- Many BTB implementations do not tag by security domain
- Prior research: Break Address Space Layout Randomization (ASLR) across security domains ("Jump over ASLR" paper)
- **Inject misspeculation to controlled addresses across security domains**
- Attack goal: Leak host memory from inside a KVM guest

Known predictor internals

"Jump over ASLR" paper on direct branch prediction:

- bits 0-30 of the source go into BTB indexing function
- BTB collisions between userspace processes are possible
- BTB collisions between userspace and kernel are possible

https://github.com/felixwilhelm/mario_baslr:

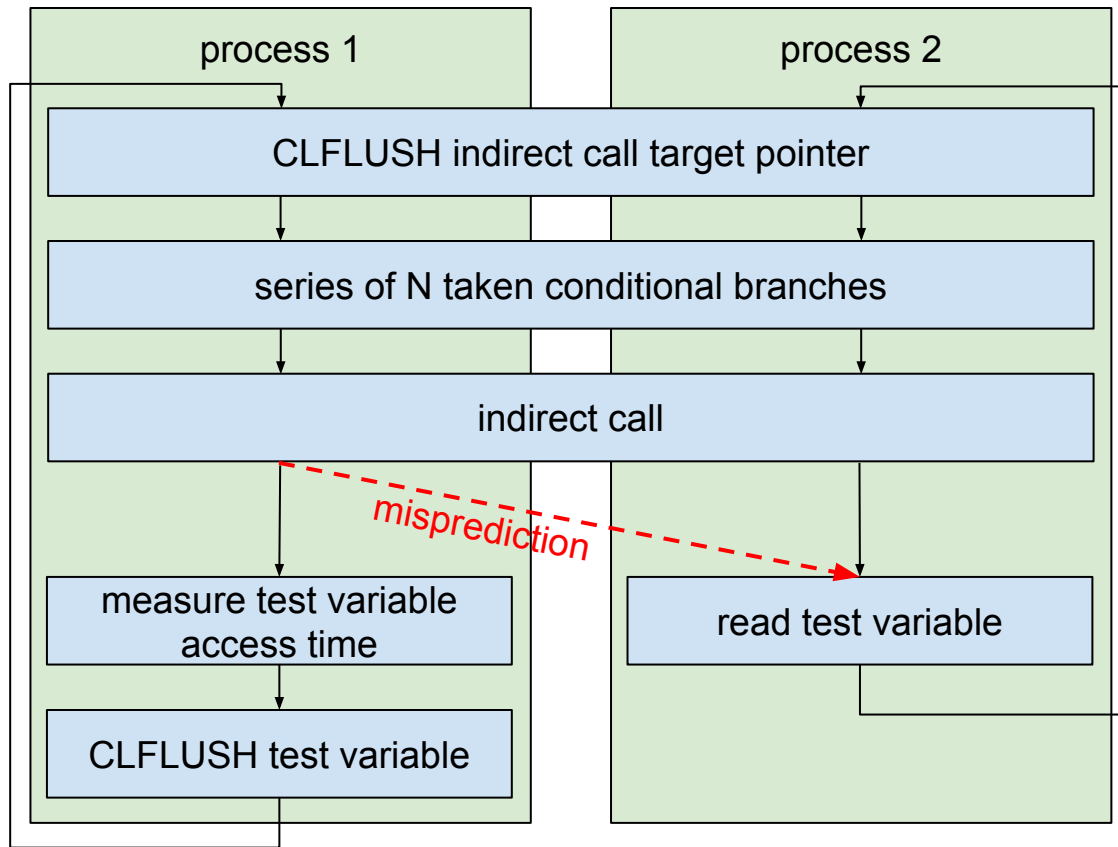
- BTB collisions between VT-x guest and host are possible

Intel Optimization Manual on Intel Core uarch:

- predictions are calculated for 32-byte blocks of source instructions
- conditional branches: predicts both taken/not taken and target address
- indirect branches: two prediction modes:
 - "monotonic target"
 - "targets that vary in accordance with recent program behavior"

Minimal Test

- run two processes in parallel
- on same physical core (hyperthreaded)
- same code
- same memory layout (no ASLR)
- different indirect call targets
- process 1: normally measures and flushes test variable in a loop
- target injection from process 2 into process 1 can cause extra load
- [explicit execution barriers omitted from diagram]



Variant 2: first brittle PoC [in initial writeup]

- minimize the problem for a minimal PoC:
 - add cheats for finding host addresses
 - add cheat for flushing host cacheline with function pointers
- use BTB structure information from prior research ("Jump over ASLR" paper)
 - Source address: low 31 bits
 - "Jump over ASLR" looked at prediction for direct branches!
- collide low 31 bits of source address, assume relative target
- leak rate: ~6 bits/second
- almost all the injection attempts fail!
- somehow the CPU can distinguish injections and hypervisor execution
- Theory:
 - injection only works for "monotonic target" prediction
 - CPU prefers history-based prediction
 - injection works when history-based prediction fails due to system noise causing evictions

Branch Prediction Model

history-based prediction

- branch source address might be used
- preceding branches are used
 - which information?
 - how many branches?
 - which kinds of branches?

reverse this sufficiently for injections?

fallback
force fallback?

"monotonic target" prediction

- uses branch source address for lookup

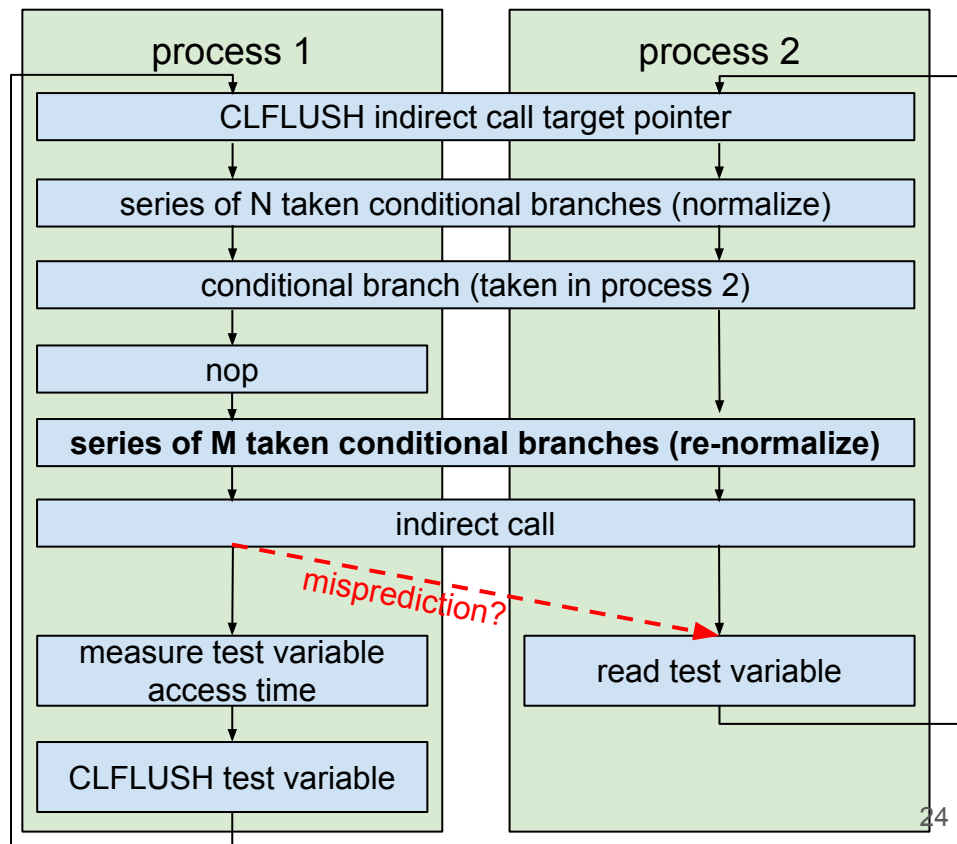
injection seems to work, but not usually used

Idea: Force predictor fallback via BTB [untested]

- poisoning the "monotonic target" predictor is relatively easy
- figure out what determines the way for the history-based predictor
- for each attack attempt:
 - flush the correct set in the history-based predictor via eviction
 - poison the "monotonic target" predictor
- good: doesn't require full knowledge about the history-based predictor
- bad: still requires knowledge of which bits are used for set selection
 - (unless you try to just spam the whole thing, which will probably break other things)
- bad: requires messing around with two predictors instead of one

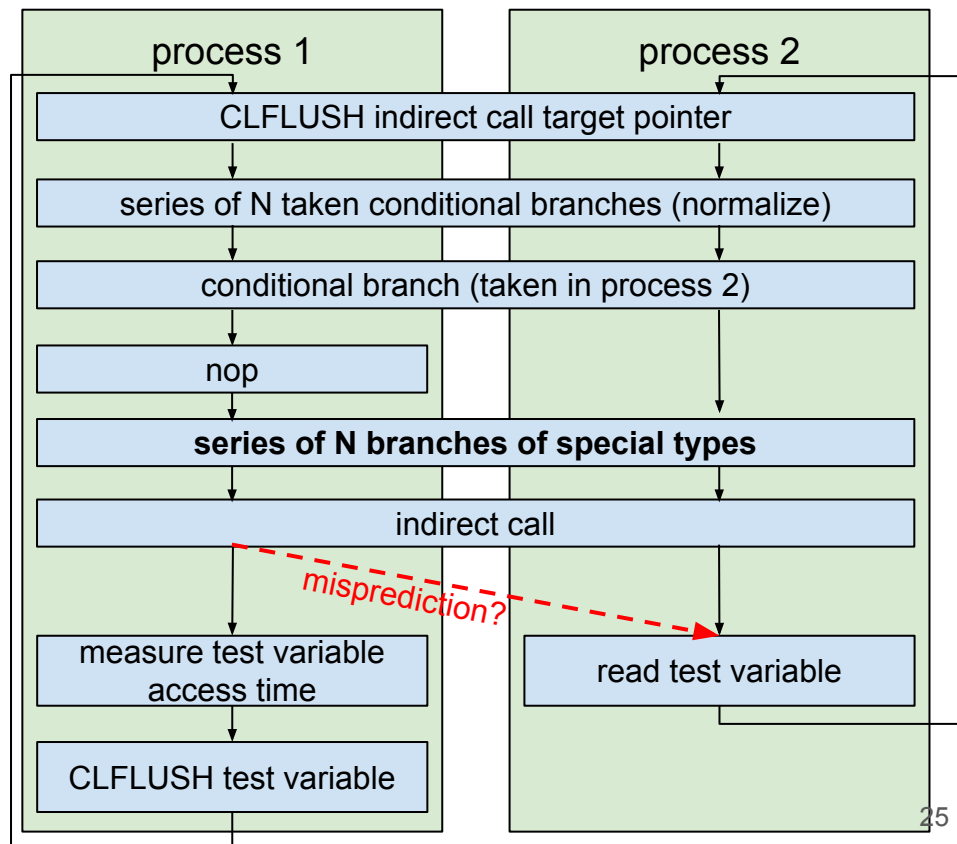
Predictor Reversing: History length

- normalize history (N taken conditional branches)
- introduce history difference (conditional branch and nop)
- attempt to re-normalize history using M branches
- measure whether injection occurred
- high injection rate indicates history collision
- result on Haswell: ~26 branches stored; but measurements get weird around the boundary [and are not yet entirely correct]



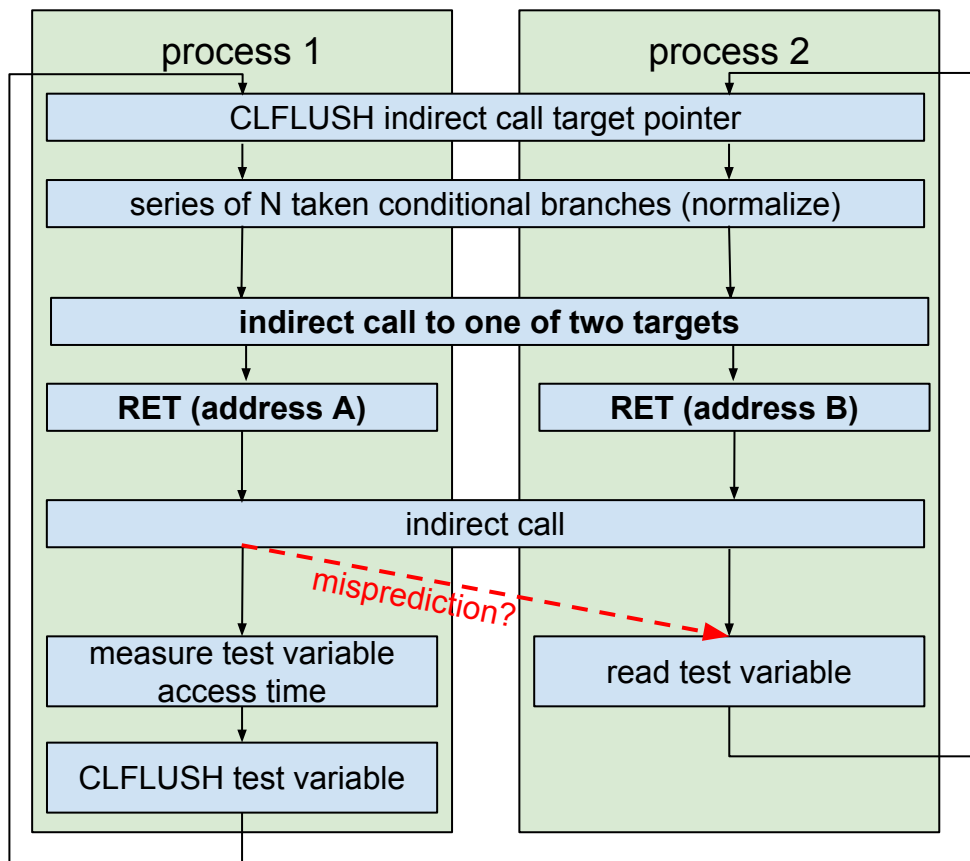
Predictor Reversing: Branch types

- results should be useful for constructing more detailed tests
- attempt to re-normalize history using N branches of a particular type
- high collision rate indicates that branches of that type don't count towards history
- on Haswell (✓ counts, ✗ doesn't):
 - taken conditional branch ✓
 - not-taken conditional branch ✗
 - unconditional direct jump ✓
 - unconditional indirect branch ✓
 - RET ✓
 - IRETQ ✗



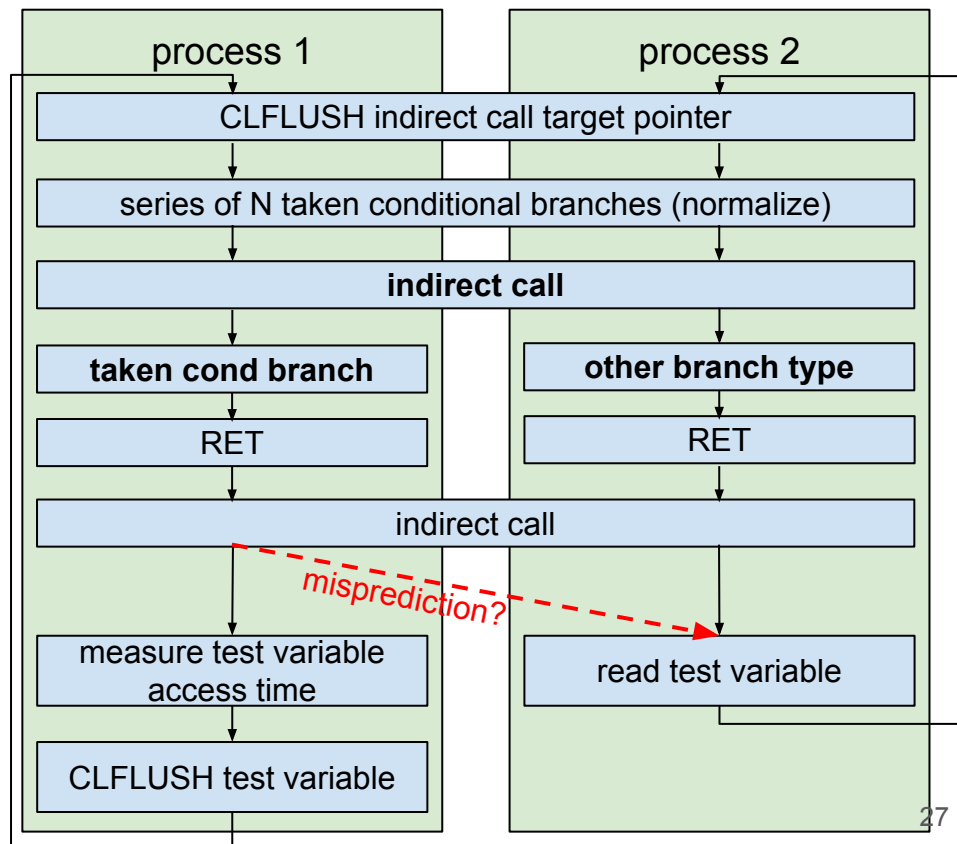
Address bits in history

- place indirect call with targets A and B before misprediction-measured call
- two sources of history difference:
 - target address of call to RET
 - source address of RET
- in multiple runs, choose A and B such that they differ in one bit each time
- result: only low 20 bits of any address affect history



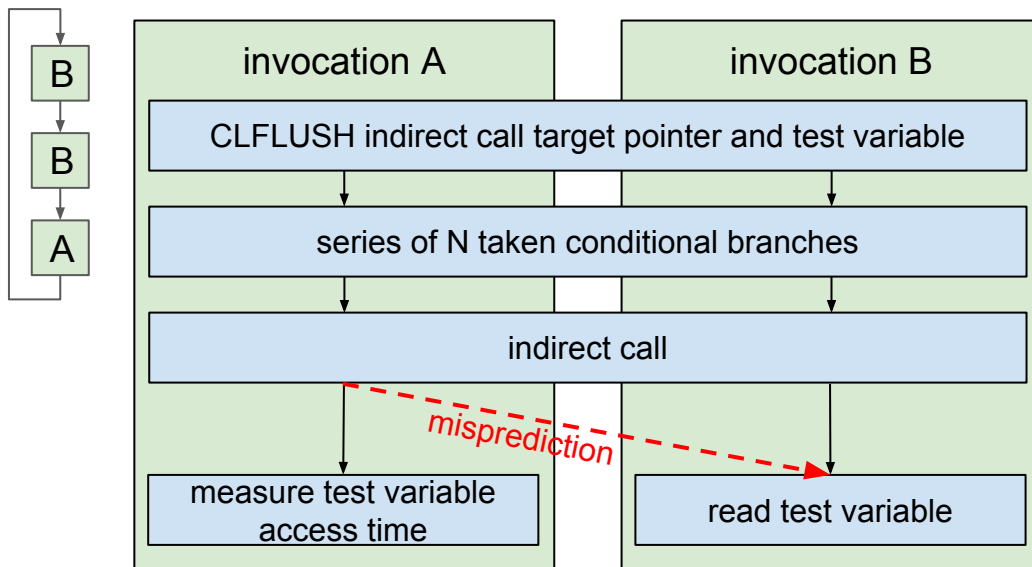
Predictor Reversing: Branch type influence?

- Does the branch type influence branch history?
 - If no, we only need to reverse the remaining history buffer details with one branch type
- Test: measure whether the CPU can distinguish execution with two different branch types in the branch history
- Pick addresses for different branch types to only differ in the high bit
- Result: Branch type doesn't matter, as long as the **last** bytes of the branch instructions are aligned



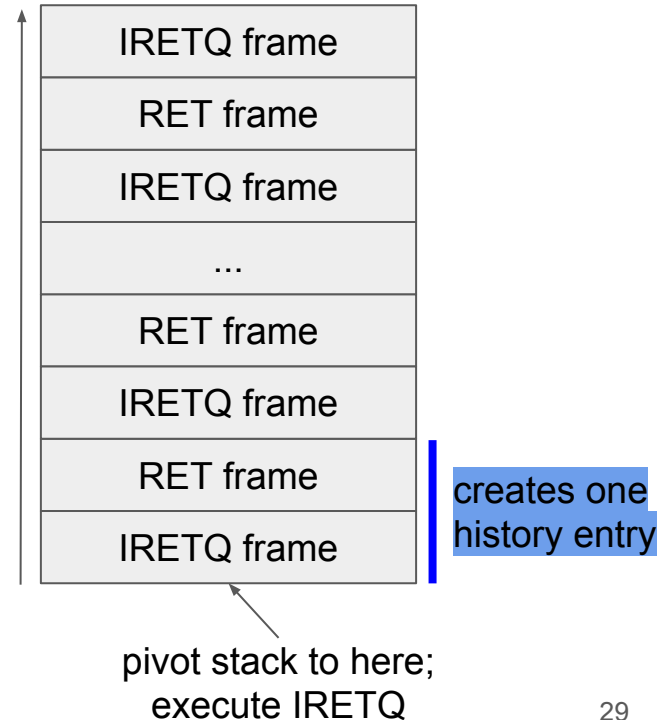
Predictor Reversing: More reliable poisoning

- single-threaded, single program
- poison twice, measure once, in a loop
- benefit: predictor poisoning should be more reliable
- downside: can't put different code at same address - but can just use aliasing addresses



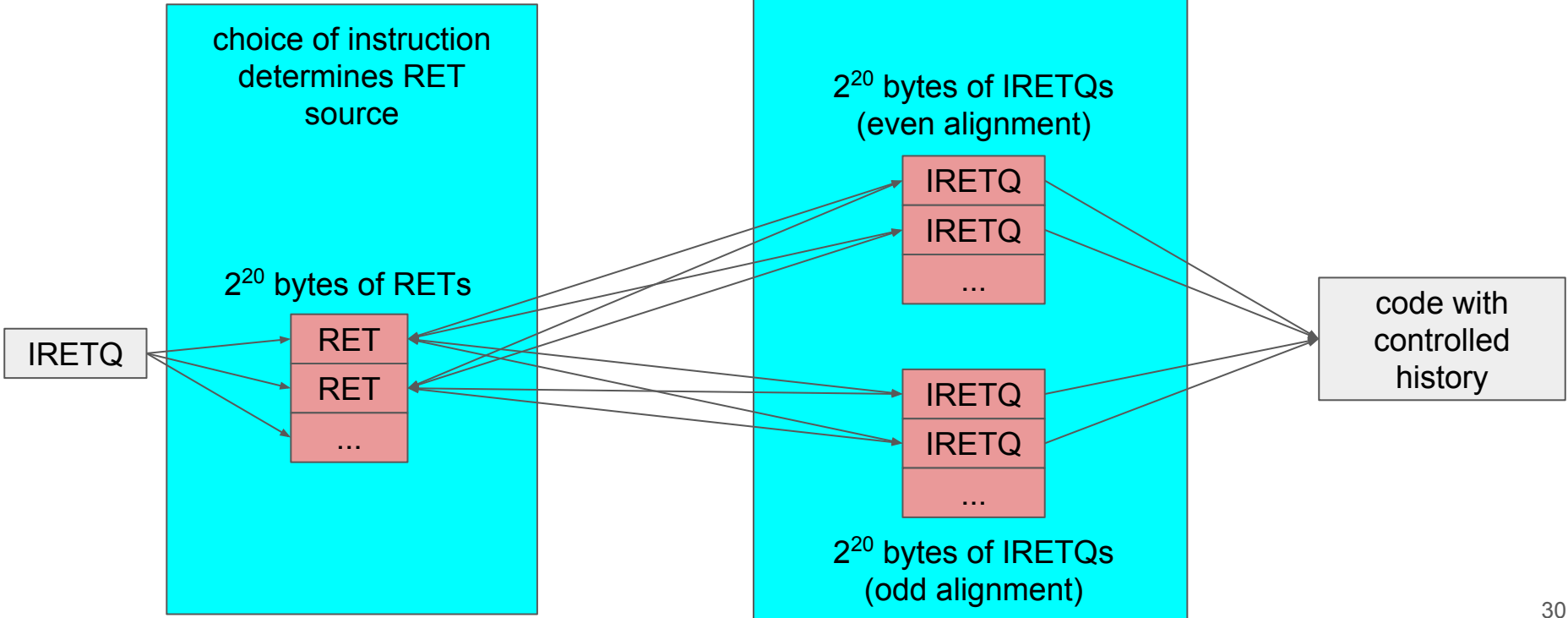
Full history control

- kinda like ROP
- use RET instructions to add history entries
 - RET reads a target from RSP, jumps to the target, and advances RSP in one byte
 - RET target is fed into predictor as **target**
 - RET target is always an IRETQ
- use IRETQ instructions to move between RET instructions
 - IRETQ target is fed into predictor as **source** (by the following RET)
 - IRETQ target, apart from the last one, is always RET



Full history control

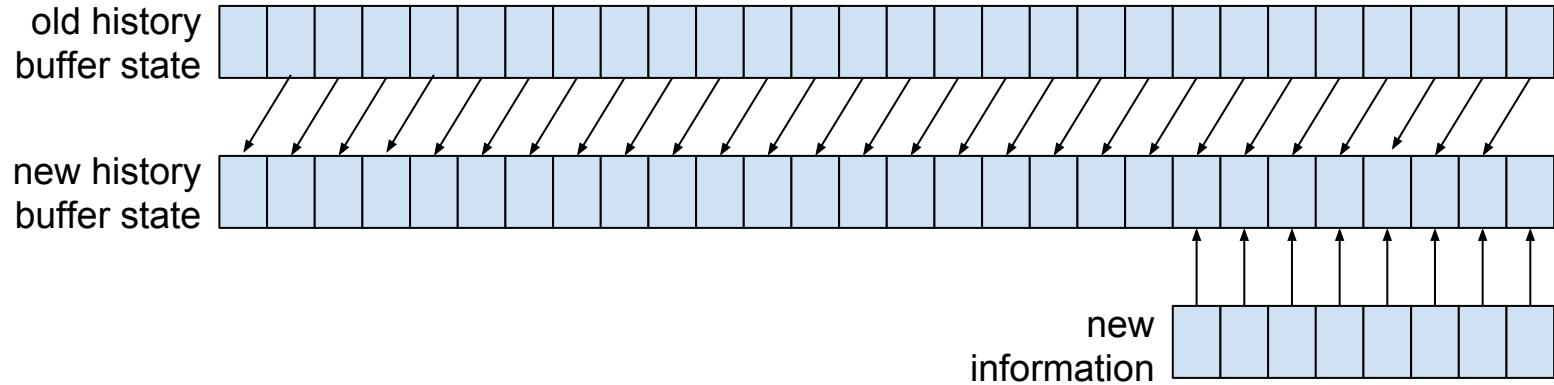
C3 RET
48CF IRETQ



History buffer structure

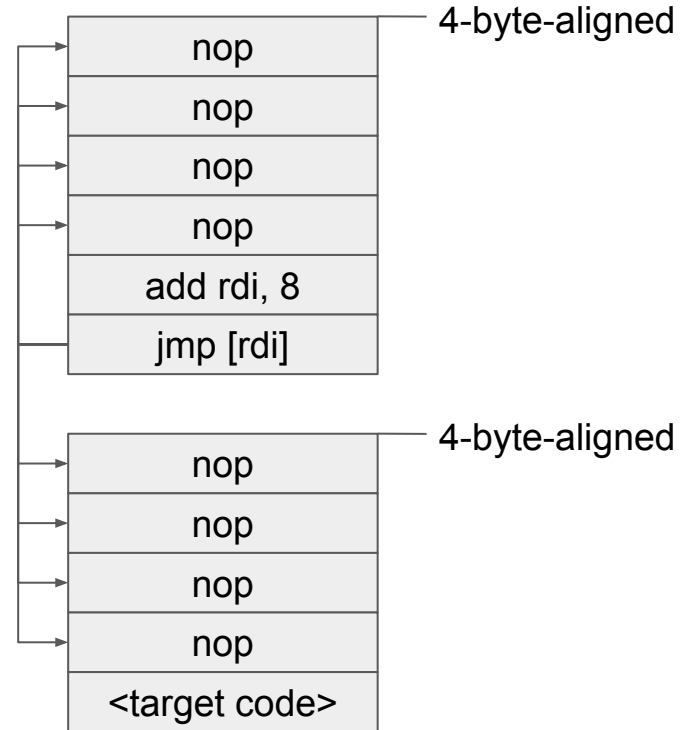
- Agner Fog's <http://agner.org/optimize/microarchitecture.pdf> describes a predictor with one bit of history (taken / not taken) per conditional branch
- good: compact storage (only one bit per history entry)
- bad: Haswell's predictor doesn't seem to store not-taken branches at all
 - must still be able to differentiate between "taken, not taken" and "not taken; taken"
 - address of taken branch is probably used
- bad: Haswell's predictor seems to be able to differentiate between many targets for a single history entry
 - but should still have compact storage!
 - history entries must be mixed together somehow
 - XOR! it's fast, and it isn't terrible at mixing data
- good: naturally forgets about old branches (shifted out)
 - data must not be propagated towards newer bits

History buffer structure



Simplified history control (untested)

- 2 bits controlled history buffer input per jump
- jumps must otherwise have constant effect on history buffer
- less IRETQs, should be faster



Attacking KVM: Overview

- goal: read from arbitrary host-kernel-virtual addresses
- attacker type: controls guest ring 0; knows precise host kernel build
- misdirect first indirect call with memory operand after guest exit
 - provides speculative RIP control
 - requires breaking hypervisor code ASLR
- flush L3 cache line containing memory operand
 - requires L3 eviction sets (for long speculation)
 - requires identifying correct eviction set
- use gadget to call into BPF interpreter
 - requires register control: caller-saved registers stay intact after guest exit
 - requires data at known address: locate host physmap alias of guest memory
- use BPF bytecode to read arbitrary host data and leak it

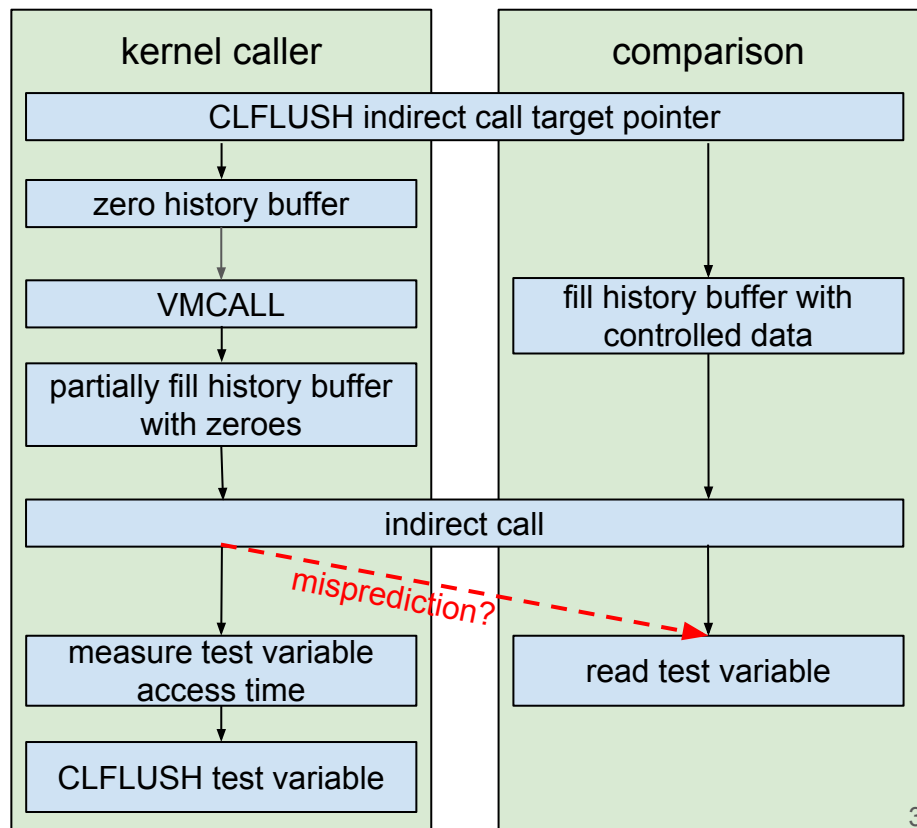
Attacking KVM: Steps overview

- leak **host code address bits** from history buffer and branch target buffer (BTB) *[dump_hyper_bhb, hyper_btb_brute]*
- identify **L3 cache sets** using brute-force timing-based testing of eviction sets *[cacheset_identify]*
- determine **physical address of guest page** using "load from physical address" gadget and timing *[find_phys_mapping_kassist]*
- determine **address of physmap region** using memory load gadget and timing *[find_page_offset]*
- select **L3 set containing the legitimate indirect call target** using brute force *[select_set]*

Leaking host address bits (BHB)

approach: dump history buffer contents

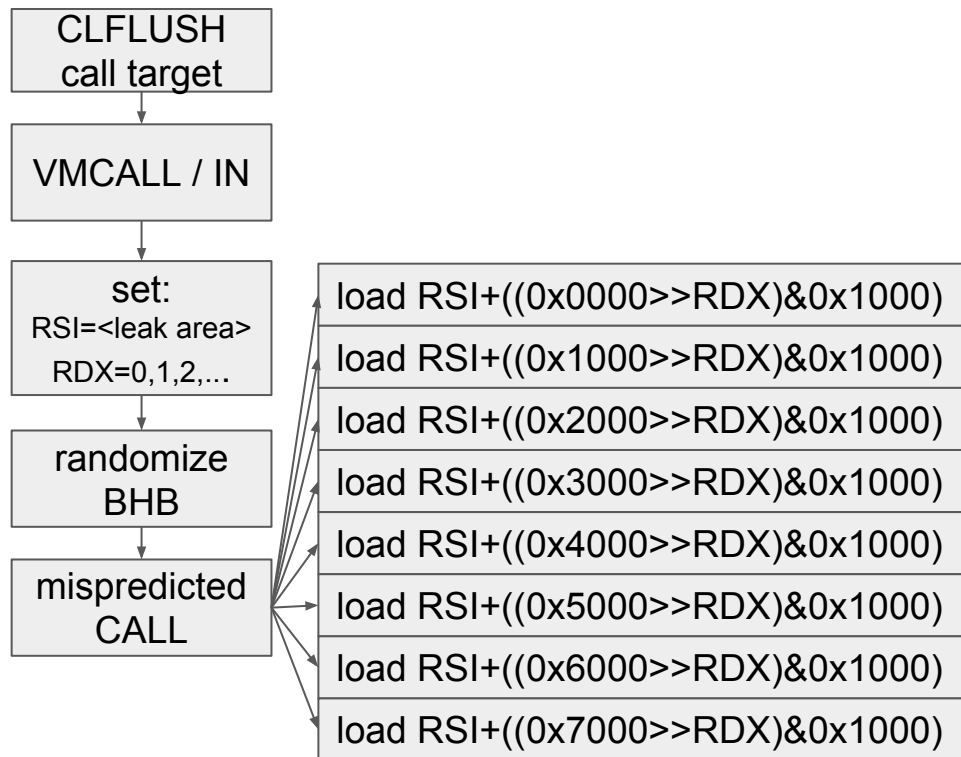
- fill history buffer with state from VMCALL
- shift out some of VMCALL state by padding history buffer with zeroes; leaving 2 bits of unknown information
- compare history buffer against controlled history buffer using misprediction



Leaking host address bits (BTB)

approach: execute an indirect call and observe where the CPU jumps

- perform VM exit (VMCALL / IN) to fill BTB with host jump addresses
- randomize history buffer to force predictor fallback
- execute CALL with mispredicted target
- place cache-signalling gadgets at all possible targets; two possible signals
- perform binary search over call targets



Identifying L3 eviction sets

Simplified algorithm:

In a loop, on a large set of cache lines with the same in-page alignment:

- choose a random set of cache lines (expected to contain one eviction set, modeled as binomial distribution)
- repeatedly remove elements from the set while checking that the set doesn't fit into the cache
 - if the set does fit, revert last change

example images: 3 sets, 2-way associative

from http://palms.ee.princeton.edu/system/files/SP_vfinal.pdf, section IV

slow set

used removable	unused		
used removable	used removable	used removable	used removable ✗
unused	unused		

used removable	unused		
used needed	used needed	used needed ✗	removed
unused	unused		

used removable	unused		
used needed	used needed	removed	removed
unused	unused		

Locate guest page in host memory

Find host-physical address:

- execute misspeculated host code using BTB poisoning and L1D+L2 (not L3!) eviction set
- Use physical-load gadget (see right) to bruteforce physical address
 - test guesses with FLUSH+RELOAD

```
// controlled r8, r9
mov    rax,r8
movsxd r15,r9d
// load page_offset_base
mov    r8,QWORD PTR [r15*8-0x7e594c40]
lea    rdi,[rax+r8*1]
// page_offset_base + phys_addr_guess
mov    r12,QWORD PTR [r8+rax*1+0xf8]
```

Find host-virtual address:

- physmap is 1GiB-aligned
- bruteforce physmap base address
- test guesses by attempting to access `page_offset_base + physical_guest_page_address`

Leak host memory

- place Spectre gadget BPF bytecode in guest memory
- flush leak area
- flush call target using L3 eviction pattern
- mistrain branch predictor to BPF interpreter call gadget
- execute VMCALL
- probe timings in leak area