A Language for Microarchitectural Security Evaluation

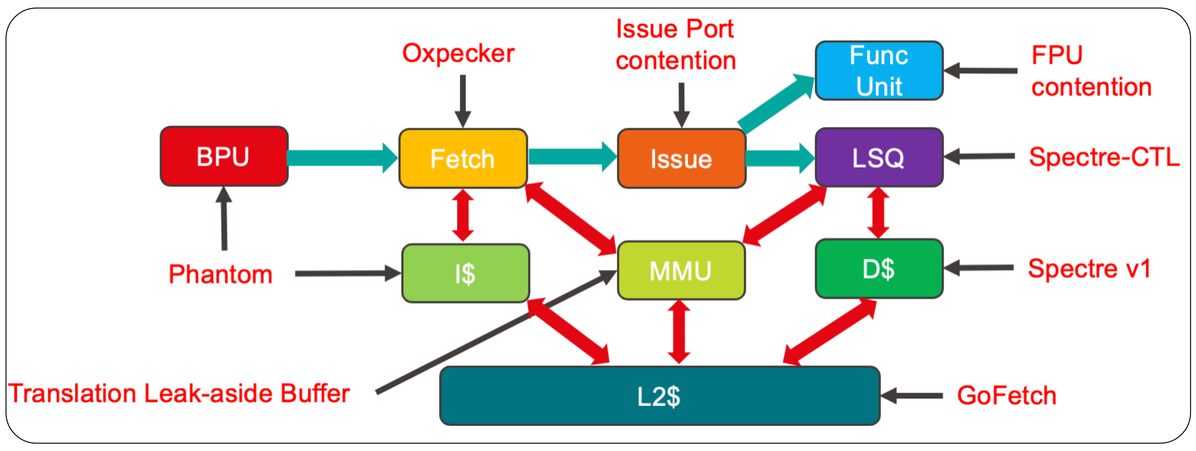
EPFL

Guokai Chen Thomas Bourgeat

École Polytechnique Fédérale de Lausanne

Introduction

Attack surfaces span modern microarchitecture



Yet still a pain to construct attack programs

- **Limited and verbose control over address layout**

```
__attribute__((aligned(4096)))
static uint64_t results1[RB_SLOTS] = {0};
```

- **Heavy usage of non-portable assembly**

```
// Inserting recursive PhantomCALL
asm("mov $1f, %%r10\n\t"
    "mov $" xstr(PHANTOM_CALL) ", %%r8\n\t"
    "mov $" xstr(CALL_FN_TRAIN_ALIAS) ", %%r9\n\t"
    "jmp *%%r9\n\t"
    "1: pop %%r9\n\t" ::: "r8", "r9", "r10");
```

- **Mixture of victim and attacker**

```
pid_t sibling_proc = run_sibling_noise("./workload", CORE2);
set_cpu_affinity(CORE1);

base = (uint64_t) aligned_alloc(HUGEPAGE, HUGEPAGE);
madvise((void *) base, HUGEPAGE, MADV_HUGEPAGE);
```

- **Lack of primitives**

```
static inline __attribute__((always_inline)) uint64_t rdtscp(void) {
    uint64_t lo, hi;
    asm volatile("RDTSCP\n\t"
                 "movq %%rdx, %0\n\t"
                 "movq %%rax, %1\n\t"
                 "CPUID\n\t"
                 : "=r"(hi), "=r"(lo)::"%rax", "%rbx", "%rcx", "%rdx");
    return (hi << 32) | lo;
}
```

- **Black boxes – No access to march design data**
- Rely on reverse engineering

Challenge and Solution

Mismatch between language and usage

- High-level language abstract away march details
- Assembly is too low-level to be easy-to-construct, portable and understandable

Solution: A language for microarchitectural security evaluation

- Highly portable
- Fine-grained layout control
- Clear victim-attacker isolation
- Enable novel open source design assisted flow

Key Features

World Isolation

```
Victim{
  Control{
    Attacker() {
    }
  }
}
```

Constraint based layout control

```
val myrule = placement[types]("myrule1")("a" -> types.Addr) {
  a("a") > imm(0x10000000)
  a("a") % imm(0x40) === imm(0)
}
ControlflowInst(Some(myrule), None, "branch1")

Jmp("myjmp", go("jmpdest"), Some(go("branch1")))
```

Description and march decoupling

No more pseudo code!

```
func[types](types.Bool)("victim")() {
  Bool("true", name = "ret")

  Padding()
  // victim should share BP history here
  Attacker() {
    FlushBPHistory()
  }
  Jmp(name = "victim_jmp",
      target = go("victim_jmp_dest"),
      inst = Some(go("victim_jmp")))

  Padding()
  PlaceLabel(go("victim_jmp_dest"))

  ret(Some(gv("ret")))
}
```

```
Intel14GJmpCollision(
  victim = go("victim_jmp"),
  attacker = go("attack_jmp")
)
```

Underlying

```
(o2p(victim) - o2p(attacker))
  % imm(0x400000) === imm(0)
```

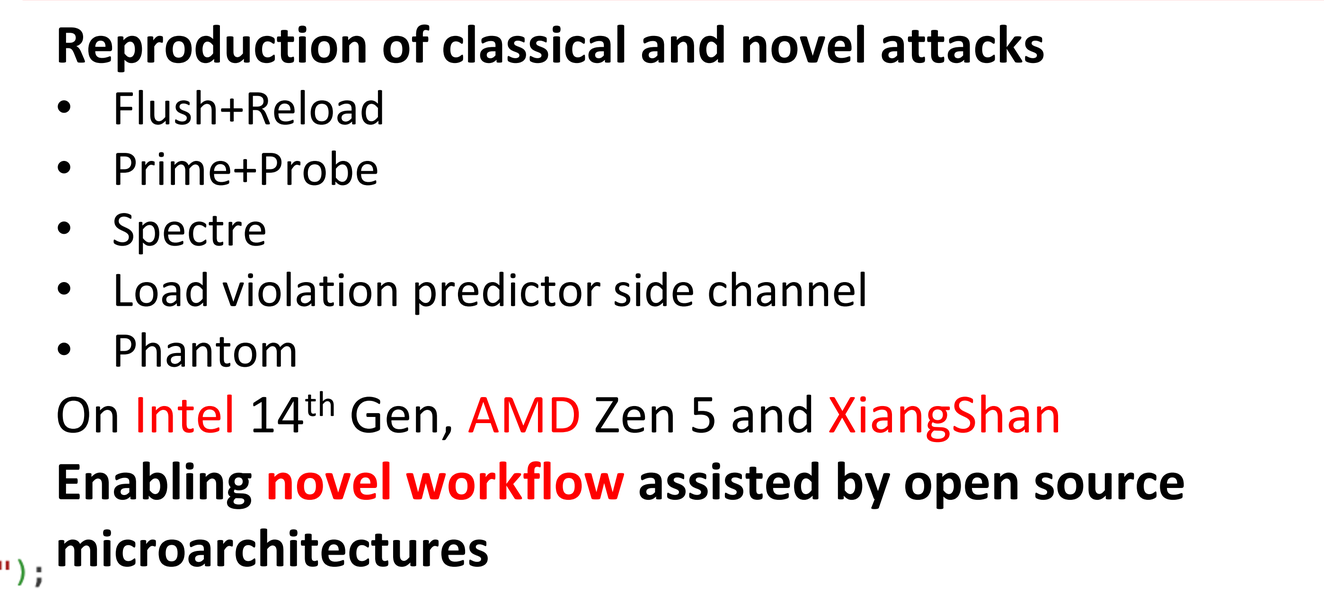
Evaluation

Reproduction of classical and novel attacks

- Flush+Reload
- Prime+Probe
- Spectre
- Load violation predictor side channel
- Phantom

On Intel 14th Gen, AMD Zen 5 and XiangShan

Enabling novel workflow assisted by open source microarchitectures



- Offers description correctness before testing on commercial platforms
- Identified a **new Phantom variant**: Phantom-coherence, enabling checking Phantom fetch from data cache side