BLUEHAT

IL 2022

An Analysis of Speculative Type Confusion Vulnerabilities in the Wild

Adam Morrison

About the speaker

Associate professor, CS, Tel Aviv University.

Started career doing vulnerability research.

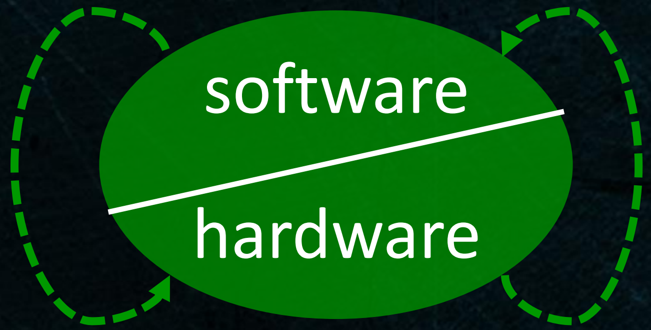
About the speaker

Associate professor, CS, Tel Aviv University.

Started career doing vulnerability research.

Now: Software/hardware interactions.

- From microarchitecture to OS.



About this talk

Joint work with **Ofek Kirzner**.

Full details available in academic paper:

An Analysis of Speculative Type Confusion Vulnerabilities in the Wild

Ofek Kirzner Adam Morrison
Tel Aviv University

Spectre attacks

THE WALL STREET JOURNAL.

TECH

Businesses Rush to Contain Fallout From Major Chip Flaws

Software patches to plug holes could slow computers, experts say

The New York Times

Researchers Discover Two Major Flaws in the World's Computers

The Washington Post

Democracy Dies in Darkness

Technology

Huge security flaws revealed — and tech companies can barely keep up

Executive summary

Spectre v1 has no hardware fix; software mitigations required.

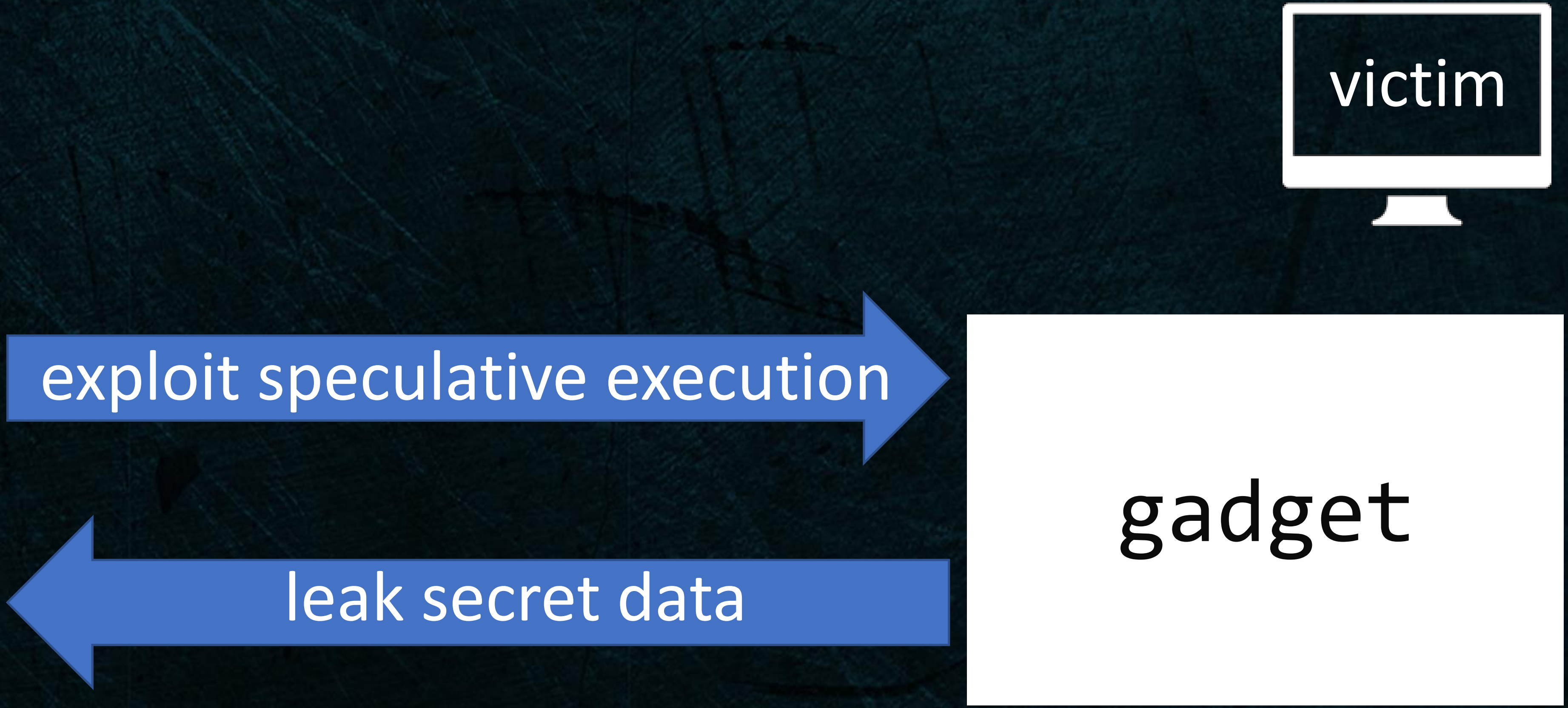
Popularized as a bounds-check bypass attack.

We show **other Spectre v1 vectors** in real code.

$\Rightarrow$ **Mitigating Spectre v1 in software requires some rethinking.**

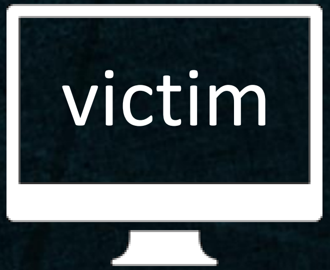
Spectre attacks

Spectre attacks



Goal: Leak data from the victim address space

Spectre variant 1: Bounds Check Bypass



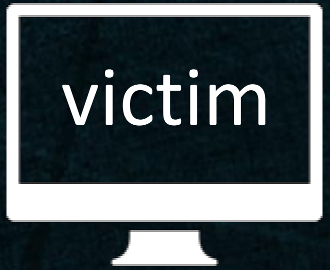
Goal: Leak data from the victim address space

exploit branch prediction

leak secret data

gadget

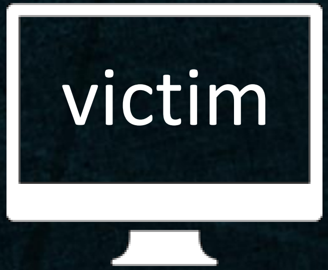
Spectre variant 1: Bounds Check Bypass



Goal: Leak data from the victim address space

```
void foo(long x) {
    // ...
    if (x < array1_len) {
        y = array1[x];
        z = array2[y * 4096];
    }
    // ...
}
```

Spectre variant 1: Bounds Check Bypass



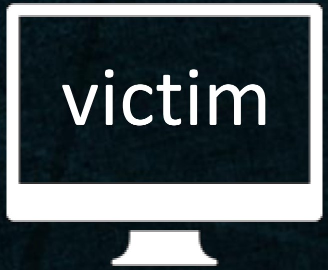
foo(x)

Goal: Leak data from the victim address space

```
void foo(long x) {
    // ...
    if (x < array1_len) {
        y = array1[x];
        z = array2[y * 4096];
    }
    // ...
}
```

Spectre variant 1: Bounds Check Bypass

speculation starts



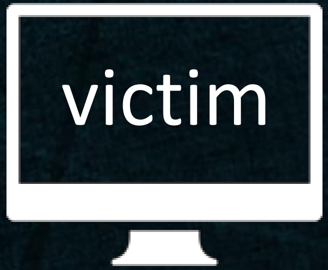
foo(&secret-array1)

Goal: Leak data from the victim address space

```
void foo(long x) {
    // ...
    if (x < array1_len) {
        y = array1[x];
        z = array2[y * 4096];
    }
    // ...
}
```

Spectre variant 1: Bounds Check Bypass

speculation starts → speculative access



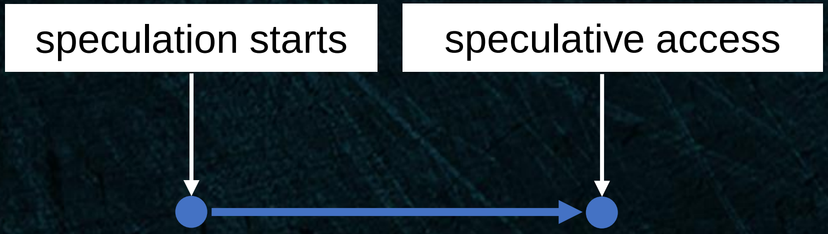
foo(&secret-array1)

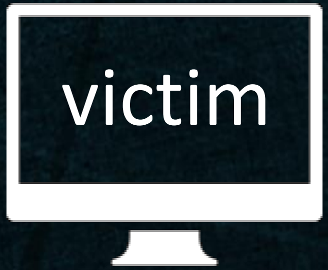
Goal: Leak data from the victim address space

array[x]=&secret

```
void foo(long x) {
    // ...
    if (x < array1_len) {
        y = array1[x];
        z = array2[y * 4096];
    }
    // ...
}
```

Spectre variant 1: Bounds Check Bypass

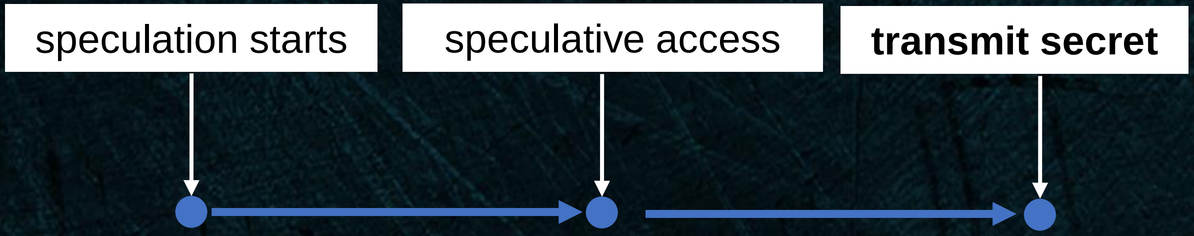


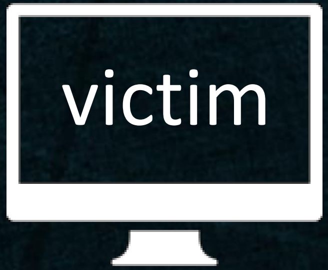


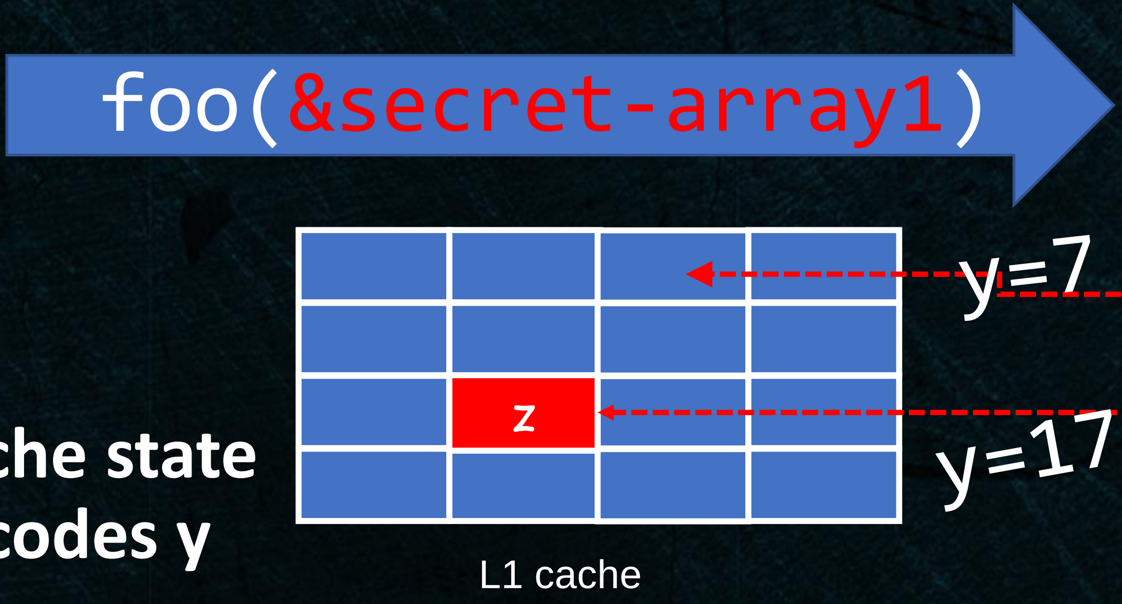
Goal: Leak data from the victim address space

```
void foo(long x) {
    // ...
    if (x < array1_len) {
        y = array1[x];
        z = array2[y * 4096];
    }
    // ...
}
```

Spectre variant 1: Bounds Check Bypass





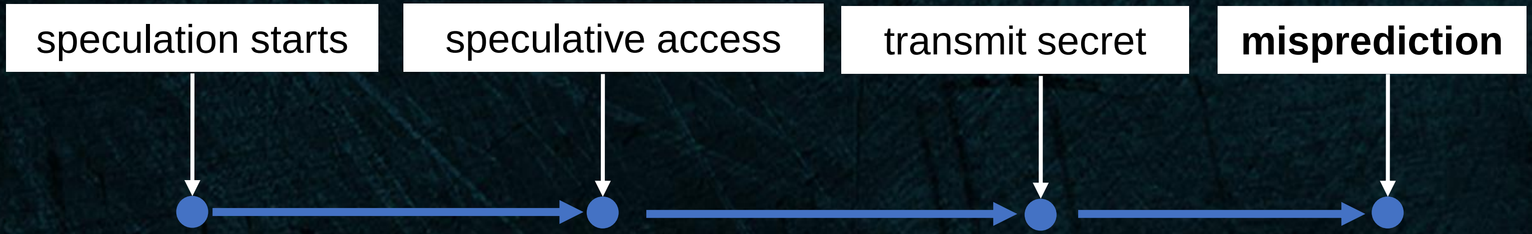


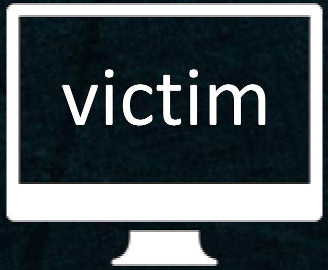

```
void foo(long x) {
    // ...
    if (x < array1_len) {
        y = array1[x];
        z = array2[y * 4096];
    }
    // ...
}
```

Goal: Leak data from the victim address space

Cache state encodes y

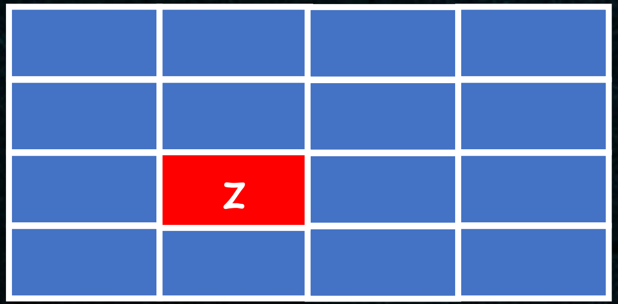
Spectre variant 1: Bounds Check Bypass





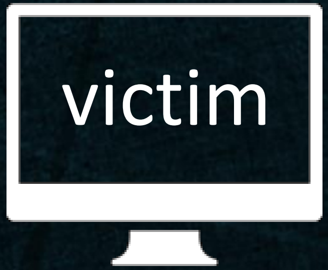
Goal: Leak data from the victim address space


```
void foo(long x) {
    // ...
    if (x < array1_len) {

    }
    // ...
}
```

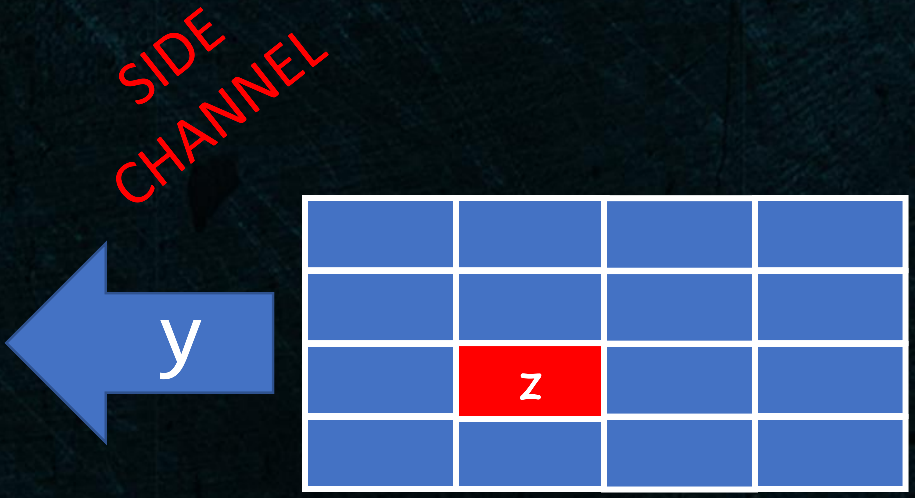


L1 cache

Spectre variant 1: Bounds Check Bypass



Goal: Leak data from the victim address space

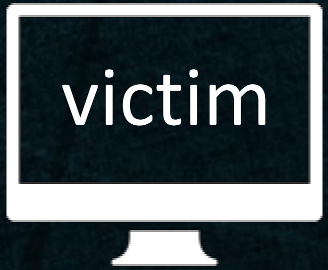


L1 cache

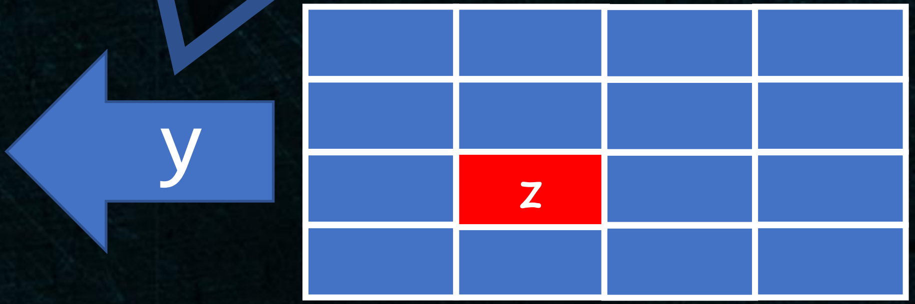
```
void foo(long x) {
    // ...
    if (x < array1_len) {
        y = array1[x];
        z = array2[y * 4096];
    }
    // ...
}
```

Spectre variant 1: Bounds Check Bypass

SIDE CHANNEL: secret-dependent microarchitectural state.



Goal: Leak data from the victim address space



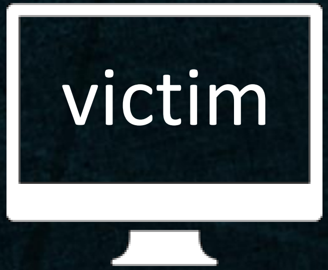
L1 cache

```
void foo(long x) {
    // ...
    if (x < array1_len) {
        y = array1[x];
        z = array2[y * 4096];
    }
    // ...
}
```

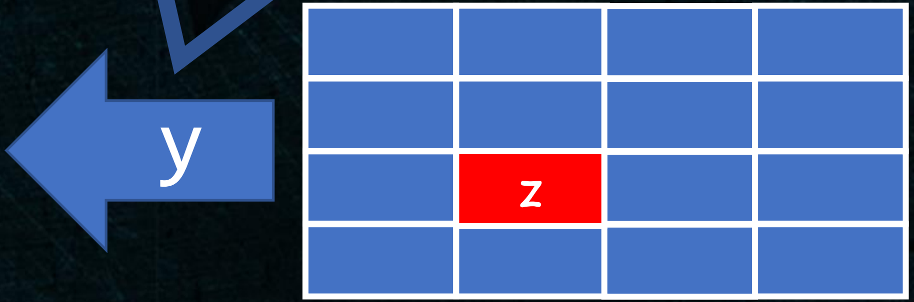
Spectre variant 1: Bounds Check Bypass

SIDE CHANNEL: secret-dependent microarchitectural state.

Spectre enabled arbitrary data to be transmitted via a side channel.



Goal: Leak data from the victim address space



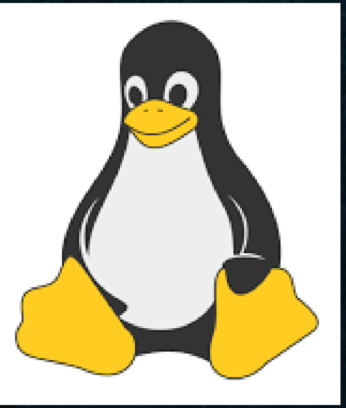
L1 cache

```
void foo(long x) {
    // ...
    if (x < array1_len) {
        y = array1[x];
        z = array2[y * 4096];
    }
    // ...
}
```

Spectre v1 is a threat to OS kernels

Attacker: unprivileged user

Attacker Victim: OS kernel



exploit system calls

Goal: Leak data from the victim address space

read any physical memory

```
void foo(long x) {
    // ...
    if (x < array1_len) {
        y = array1[x];
        z = array2[y * 4096];
    }
    // ...
}
```

Spectre v1 mitigation

No hardware fix for Spectre v1

intel newsroom

While Variant 1 will continue to be addressed via software mitigations,

AMD

For all flavors of variant 1, the AMD mitigation recommendation is software only

Spectre v1 mitigation in the Linux kernel

No hardware fix for Spectre v1

⇒ Linux has a special API to ensure bounds checks are respected under speculation

```c
void function_called_from_syscall(long x) {
    // ...
    if (x < array1_len) {
        y = array1[x];
        z = array2[y * 4096];
    }
    // ...
}
```

```c
void function_called_from_syscall(long x) {
    // ...
    if (x < array1_len) {
        y = array_index_nospec(array1[x], array1_len);
        z = array2[y * 4096];
    }
    // ...
}
```

Spectre v1: not only a bounds check bypass

Quoting from the Spectre paper [Kocher et al., 2019]:

> **Variant 1: Exploiting Conditional Branches.** In this variant of Spectre attacks, the attacker mistrains the CPU's branch predictor into mispredicting the direction of a branch, causing the CPU to temporarily violate program semantics by executing code that would not have been executed otherwise.

SPECULATIVE TYPE CONFUSION

Misspeculation makes the victim execute with some variables holding values of the wrong type, and thereby leak memory content

Speculative type confusion: example

Misprediction:
Type confusion

obj
TYPE2
…
secret

```c
void syscall_helper(struct Base* obj) {
    if (obj->type == TYPE1) {
        struct Type1* o = (struct Type1*) obj;
        leak(o->value);
    }
    if (obj->type == TYPE2) {
        ...
    }
}
```

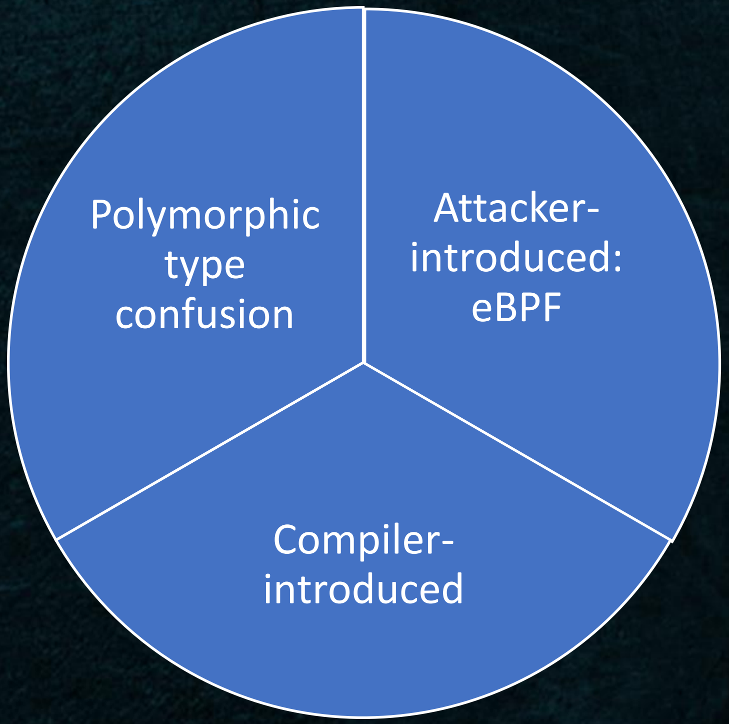
```c
struct Base {
    enum Type type;
};

struct Type1 {
    struct Base base;
    ...
    uint32_t value;
};
```

Our results

Observation: speculative type confusion may be much more prevalent than previously hypothesized.

We analyzed the Linux kernel, looking for speculative type confusion.

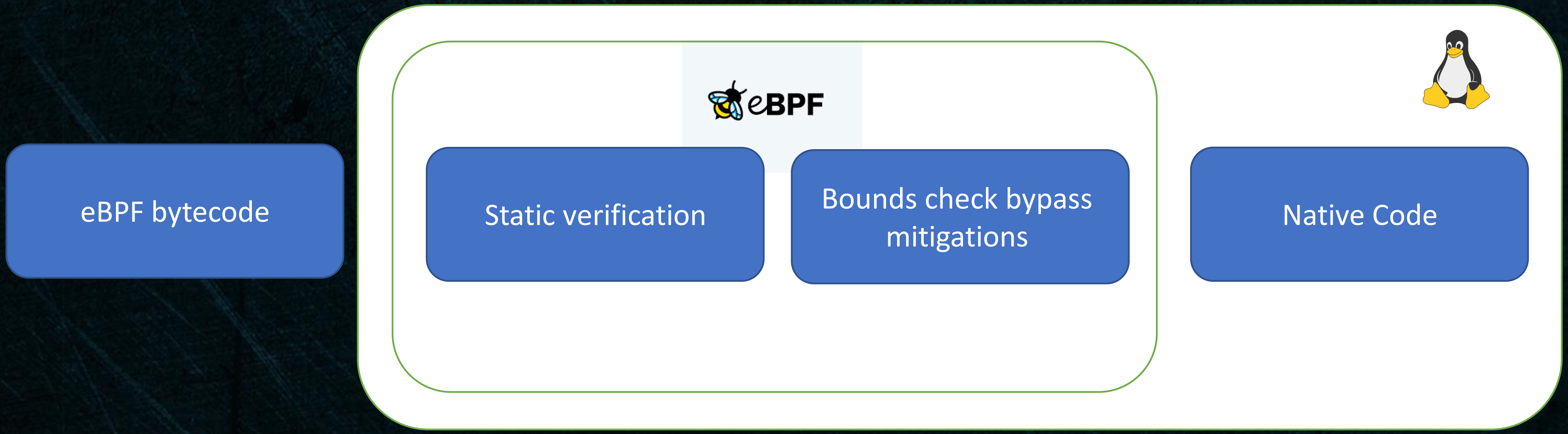
Found new types of speculative type confusion.



eBPF: Speculative Type Confusion

eBPF

Linux subsystem, enabling user-defined programs in kernel.



eBPF verifier vulnerability

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
C:     r1 = M[(r9 & 1) * 512]
D:...
```

Flows considered by eBPF verifier

r0 == 0

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
C:     r1 = M[(r9 & 1) * 512]
D:...
```

r0 == 1

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
C:     r1 = M[(r9 & 1) * 512]
D:...
```

otherwise

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
C:     r1 = M[(r9 & 1) * 512]
D:...
```

eBPF verifier vulnerability

Speculative flows are not verified

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
        r6 = r9
B: if r0 != 0x1 goto D
        r9 = *(u8 *)(r6)
C:      r1 = M[(r9 & 1) * 512]
D:...
```

Predicted “false”

Predicted “false”

```
// r0 = ptr to an array entry (verified != NULL)
// r6 = ptr to stack slot (verified != NULL)
// r9 = scalar value controlled by attacker
```

```
if r0 == 0x0 and r0 == 0x1
    r6 = r9
    r9 = *(u8 *)(r6)
    r1 = M[(r9 & 1) * 512]
```

read arbitrary memory

Training mutually exclusive branches

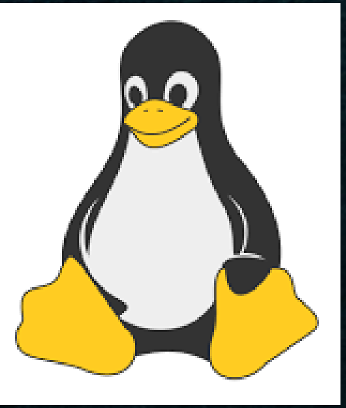
CPU

Branch Predictor

Can both be false

```
A: if r0 != 0x0 goto B
        r6 = r9
B: if r0 != 0x0 goto
        r9 = *(u8 *)
```

Shadow gadge



Mutually exclusive

```
A: if r0 != 0x0 goto B
        r6 = r9
   r0 != 0x1 goto D
        r9 = *(u8 *)(r6)
```

Unprivileged process can read arbitrary memory addresses at a rate of ~6.5 KB/sec

Compiler Introduced Speculative Type Confusion

Compilers might create speculative type confusion

Innocent looking code is compiled in a way that inroduces vulnerability

Compiler reasoning:
Branches are mutually exclusive

(trusted) ptr argument held in x86 register %rsi

attacker-controlled

```
void syscall_helper(cmd_t* cmd, char* ptr, long x)
{
    cmd_t c = *cmd;
    if (c == CMD_A)
    {
        ...
    }
    if (c == CMD_B)
    {
        y = *ptr; // y = *%rsi
        z = array[y * 4096];
    }
    // ...
}
```

code during which x moves to %rsi

Can we find it in the wild?

Binary level analysis of Linux.

Focused on system calls, which have well-defined user-controlled interface.

Leakage mechanism is out of scope: aiming at finding speculative attacker-controlled memory dereference.

compiler	flags	# vulnerable syscalls
GCC 9.3.0	-Os	20
GCC 9.3.0	-O3	2
GCC 5.8.2	-Os	0
GCC 5.8.2	-O3	0

Reusing registers for a function call

```
syscall(foo_t* uptr) {
  foo_t kfoo;
  // some code
  if (uptr)
    copy_from_user(&kfoo,
                   uptr
                   ...);
  f(uptr ? &kfoo : NULL);
  // rest of code
}
```

rdi != 0

⇒ rdi = stack var

rdi == 0

⇒ reuse rdi

Reusing registers for a function call

```
syscall(foo_t* uptr) {
  foo_t kfoo;
  // some code
  if (uptr)
    copy_from_user(&kfoo,
                   uptr,
                   ...);
  f(uptr ? &kfoo : NULL);
  // rest of code
}
```

```
# args: uptr in %rdi
  ...
  testq %rdi, %rdi
  je L # jump if %rdi==0
  # set copy_from_user args
  ...
  # %rsp contains addr of
  # stack buffer
  mov %rsp, %rdi
  call copy_from_user
L:callq f
```

rdi != 0

⇒ rdi = stack var

rdi == 0

⇒ reuse rdi

Stack slot reuse

Allocated with 1-byte opcode rather than by subtraction (4-bytes), by chance contains user-controlled value

Out param

```
long keyctl_instantiate_key_common(key_serial_t id,
                                   struct iov_iter *from,
                                   key_serial_t ringid) {
  struct key *dest_keyring;
  // ... code ...
  ret = get_instantiation_keyring(ringid,rka,&dest_keyring);
  if (ret < 0)
   goto error2;
  ret = key_instantiate_and_link(rka->target_key, payload,
                                 plen, dest_keyring,
                                 instkey);
  // above call dereferences dest_keyring
}
```

```
# %rcx is a live register from caller
push  %rcx
# ... code ...
lea   0x18(%r14),%rsi   # rka argument
mov   %rsp,%rdx         # &dest_keyring argument
mov   %r15d,%edi        # ringid argument
callq get_instantiation_keyring  # returns error
test  %rax,%rax         # if (ret < 0)
mov   %rax,%rbx
js    error2            # mispredict no error
...
mov   (%rsp),%rcx       # dest_keyring argument
# dest_keyring could be old %rcx if not
# overwritten in get_instantiation_keyring()
callq key_instantiate_and_link
```

Small change → exploitable

Hard to reason about

Loads value from stack

Overwrites out param in any case

Speculative Polymorphic Type Confusion

Spectre v2 mitigations

Spectre v2 exploits misprediction of indirect branch target addresses

Retpolines: block indirect branch prediction

Optimization: restrict speculation to **valid targets** [Linux, Amit et al., 2019]

Might create speculative type confusion vulnerabilities

Indirect branch

```
jmp *%rax
```

Direct branch to retpoline thunk

Jump to correct destination

```
# %rax = branch target
cmp $0xXXXXXXXX, %rax # target1?
jz $0xXXXXXXXX
cmp $0xYYYYYYYY, %rax # target2?
jz $0xYYYYYYYY
...
jmp ${fallback} # jmp to retpoline thunk
```

Speculative polymorphic type confusion

```
struct Common { void (*foo) (void*); };
struct A { struct Common common; char* ptr; };
struct B { struct Common common; long user_controlled_scalar; };
```

```
void some_code_path(struct Common* common) {
    /* ... */
    common->foo(common);
}
```

```
void foo_A(struct Common* common) {
    char x = *((struct A*) common)->ptr;
    leak(x);
}
```

```
foo_B()
```

Speculative polymorphic type confusion

```
struct Common { void (*foo) (void*); };
struct A { struct Common common; char* ptr; };
struct B { struct Common common; long user_controlled_scalar; };
```

```
void some_code_path(struct Common* common) {
    /* ... */
    common->foo(common);
}
```

B→user_controlled_scalar

```
void foo_A(struct Common* common) {
    char x = *((struct A*) common)->ptr;
    leak(x);
}
```

misprediction

```
# %rax = branch target
cmp $0xXXXXXXXX, %rax # target1?
jz $0xXXXXXXXX
cmp $0xYYYYYYYY, %rax # target2?
jz $0xYYYYYYYY
...
jmp ${fallback} # jmp to retpoline thunk
```

foo_B()

Analysis

Analysis

- Linux code analysis - looking at ways in which polymorphism can lead to speculative type confusion

Analysis

Analysis

- Linux code analysis - looking at ways in which polymorphism can lead to speculative type confusion

Results

- Flagged potentially vulnerable: 1000s
- ”Array indexing” instances: 100s
- All – not exploitable(?) E.g., limited user value control

Results example

```
# %rax = branch target
cmp $0xXXXXXXXX, %rax # target1?
jz $0xXXXXXXXX
cmp $0xYYYYYYYY, %rax # target2?
jz $0xYYYYYYYY
...
jmp ${fallback} # jmp to retpoline thunk
```

... max_freq values are limited

```
ssize_t max_freq_store(struct device *dev,
                       struct device_attribute *attr,
                       const char *buf, size_t count) {
 ...
 if (freq_table[0] < freq_table[df->profile->max_state - 1])
  value = freq_table[df->profile->max_state - 1];
 else
  value = freq_table[0];
 // value is stored into max_freq
}
```

Mispredicted target

```
ssize_t port_type_show(struct device *dev,
                       struct device_attribute *attr,
                       char *buf) {
 // container_of use
 struct typec_port *port = to_typec_port(dev);
 if (port->cap->type == TYPEC_PORT_DRP)
  return ...;
 return sprintf(buf, "[%s]\n",
          typec_port_power_roles[port->cap->type]);
}
```

Actual target

```
ssize_t max_freq_show(struct device *dev,
                      struct device_attribute *attr,
                      char *buf) {
 // container_of use
 struct devfreq *df = to_devfreq(dev);
 return sprintf(buf, "%lu\n", min(df->scaling_max_freq,
                                  df->max_freq));
}
```

Analysis

Analysis

- Linux code analysis - looking at ways in which polymorphism can lead to speculative type confusion

Results

- Thousands - flagged potentially vulnerable
- Hundreds - ”array indexing” instances
- All - limited speculation window or limited control on user value

Conclusion

- **Were a conditional branch-based mitigation used instead of retpolines, the kernel’s security would be on shaky ground**

Limitations / future work

Our analyses are PoC.

Not exhaustive, have false positives, and false negatives.

Much room for improvement!

$\Rightarrow$More vulnerabilities.

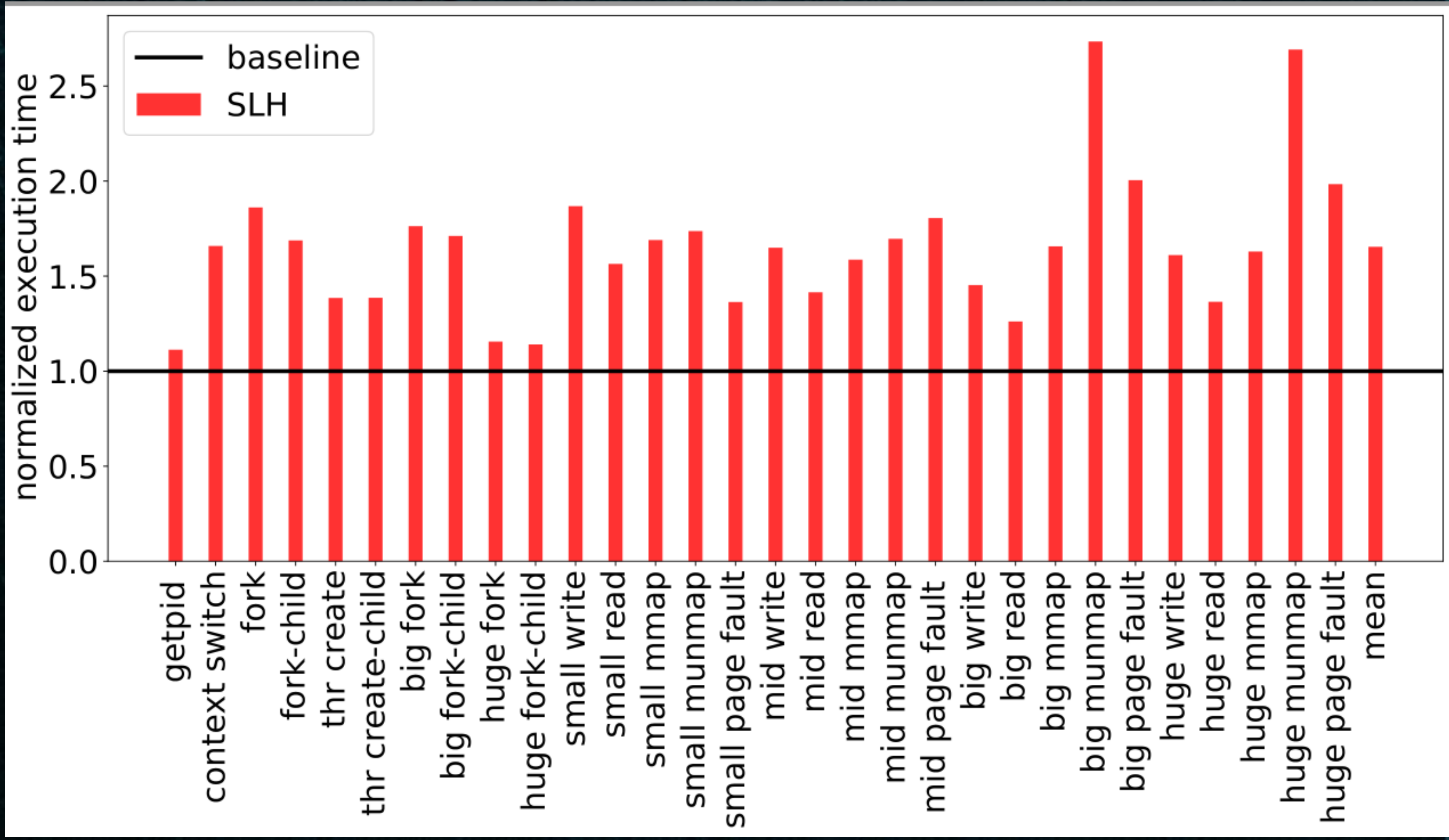
Spectre v1 software mitigation

~~Spot (manual, Linux-style)~~

Complete (compiler-based)
E.g.: LLVM SLH

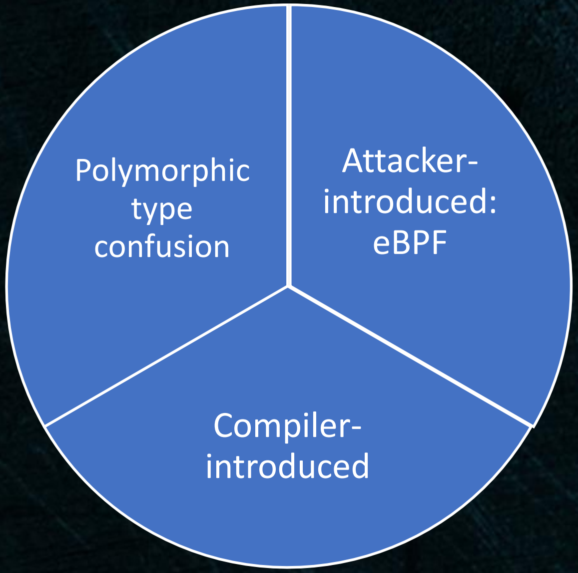
Hardware support might be required?

SLH overhead on Linux syscalls



Summary

Analysis



Conclusion

Speculative type confusion is prevalent, hard to detect

Takeaways

More bugs

Rethink Spectre v1 mitigations