Speculative Interference Attacks: Breaking Invisible Speculation Schemes

Written by Mohammad Behnia et al.
Presented by Travis Ziegler

Guiding Questions

If the order in which instructions get executed is secret dependent, how can the secret be leaked?

Is speculative execution inevitably insecure?

Problem and Motivation

Initial Problem

Spectre v1 – branch misprediction leaks data through cache accesses

```
if (i < N) {                    // speculative misprediction
    secret = A[i];
    k = B[secret*64];  // causes cachline eviction that can be noticed by attacker
}
```

Problem and Motivation

One proposed hardware solution to Spectre v1: Delay on Miss (DOM)

Make speculative cache accesses “invisible”

```
if (i < N) {                    // speculative misprediction
    secret = A[i];
    k = B[secret*64];  // On cache hit, fetch B[secret*64] and continue
                       // On cache miss, wait for speculation to resolve
}
```

Problem and Motivation

Benefits of DOM

- Allegedly fixes spectre v1 style attacks
- Does not compromise performance that much
    - Common case = contents inside speculated branch are in cache

Challenge:
How to get around DOM?

Proposal

Despite delaying cache misses during speculation, the claim is that **cache states can still get changed**!

… Which can leak secrets!

Proposal - Big Picture Idea

Pseudo-code

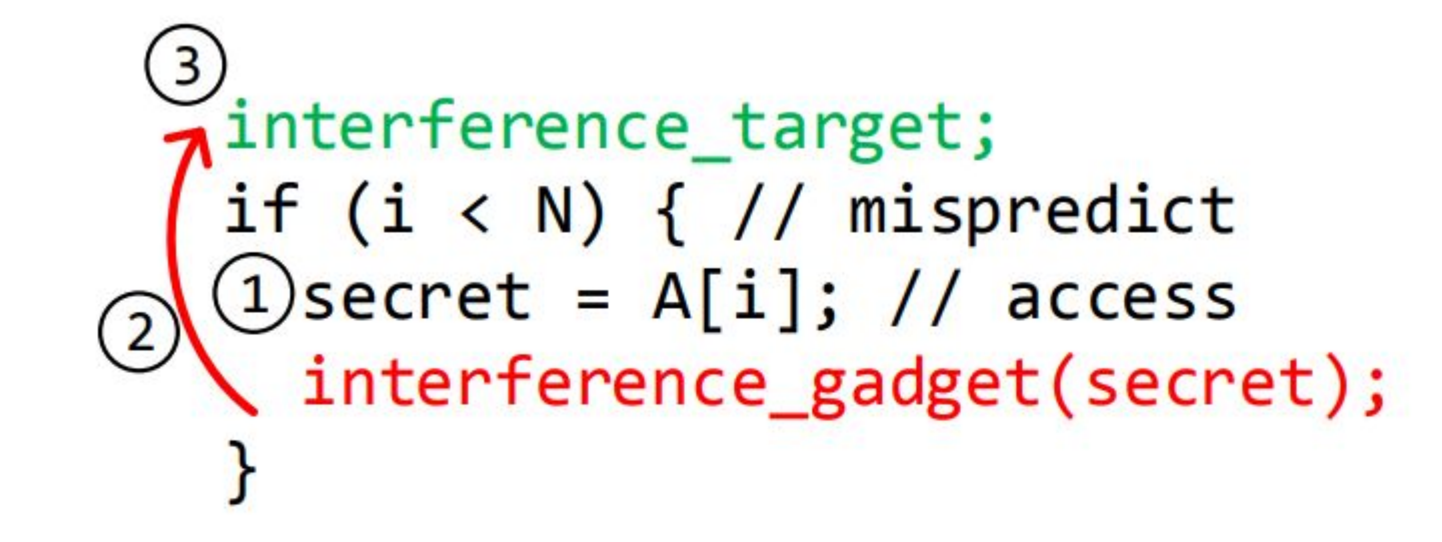
```
load(X)
load(Y)
If (i < N) {  //mispeculation }
```

Execution Order influenced by secret:

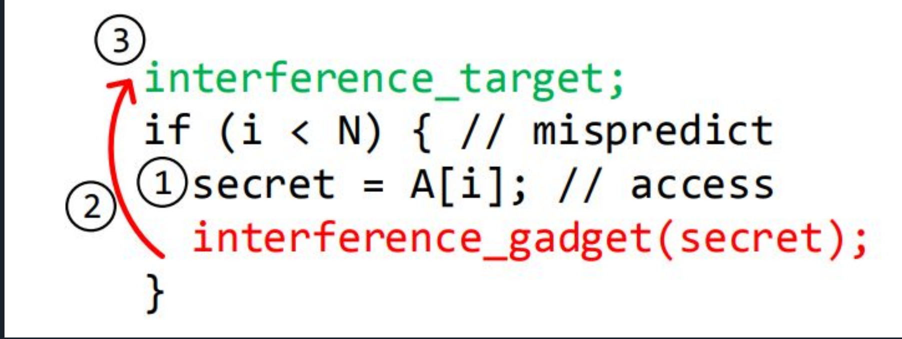
```
load(X), load(Y)     // if secret = 0

load(Y), load(X)     // if secret = 1
```

General Idea

Attack Framework to cause interference_target to get delayed



Proposal



Resource **contention** causes

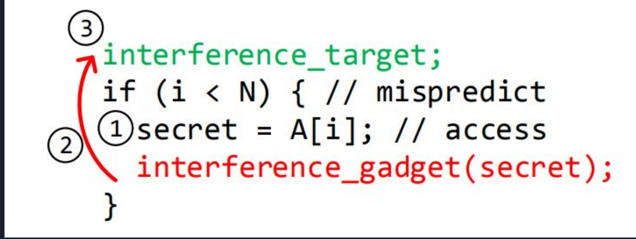
→ difference in **timing** during mis-speculation

→ difference in non-speculative instruction **execution order**

→ difference in **cache state,**

→ a **cache side channel**

Types of Interference



Interference in MSHR

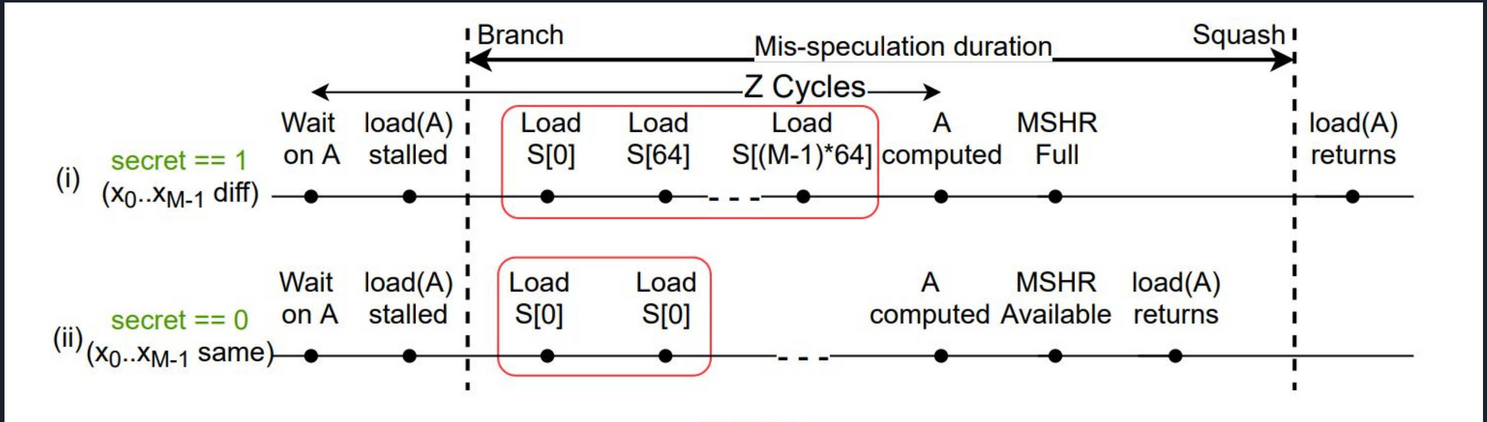
MSHR = registers needed for ongoing loads

Interference Target

A = long computation (takes Z cycles)
X = load(A)

Interference Gadget

load(&S[secret*64])
load(&S[secret*64*1])
…
load(&S[secret*64*(M-1)])



(b) (b)

```
③ interference_target;
if (i < N) { // mispredict
② ①secret = A[i]; // access
   interference_gadget(secret);
}
```

Interference in Execution Units

f(k) and f’(k) are a set of instructions that depend on k and run on the same execution unit

Interference Target

z = long computation (takes Z cycles)

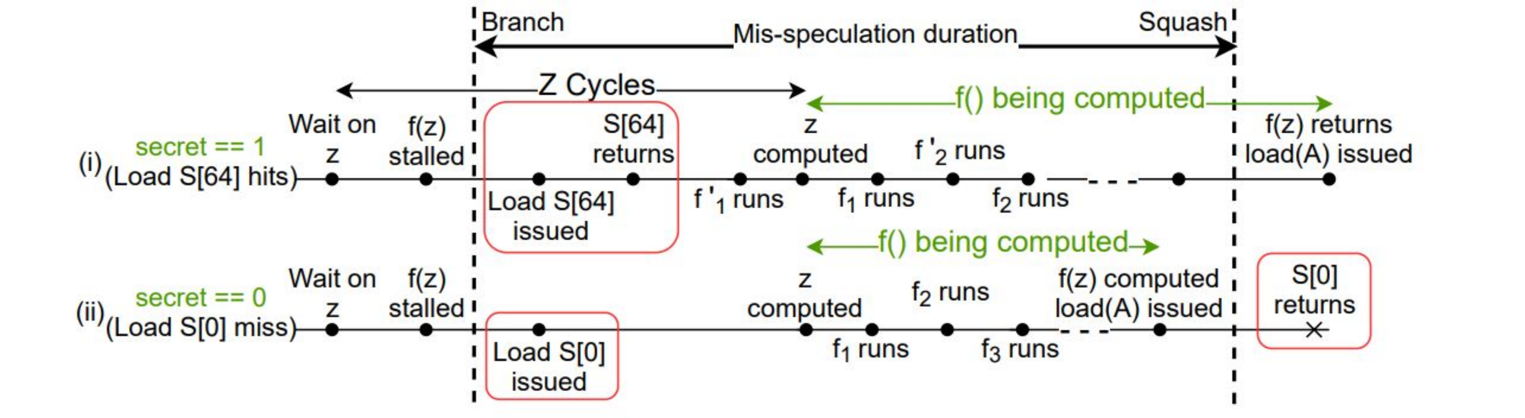
A = f(z)

X = load(A)

Interference Gadget

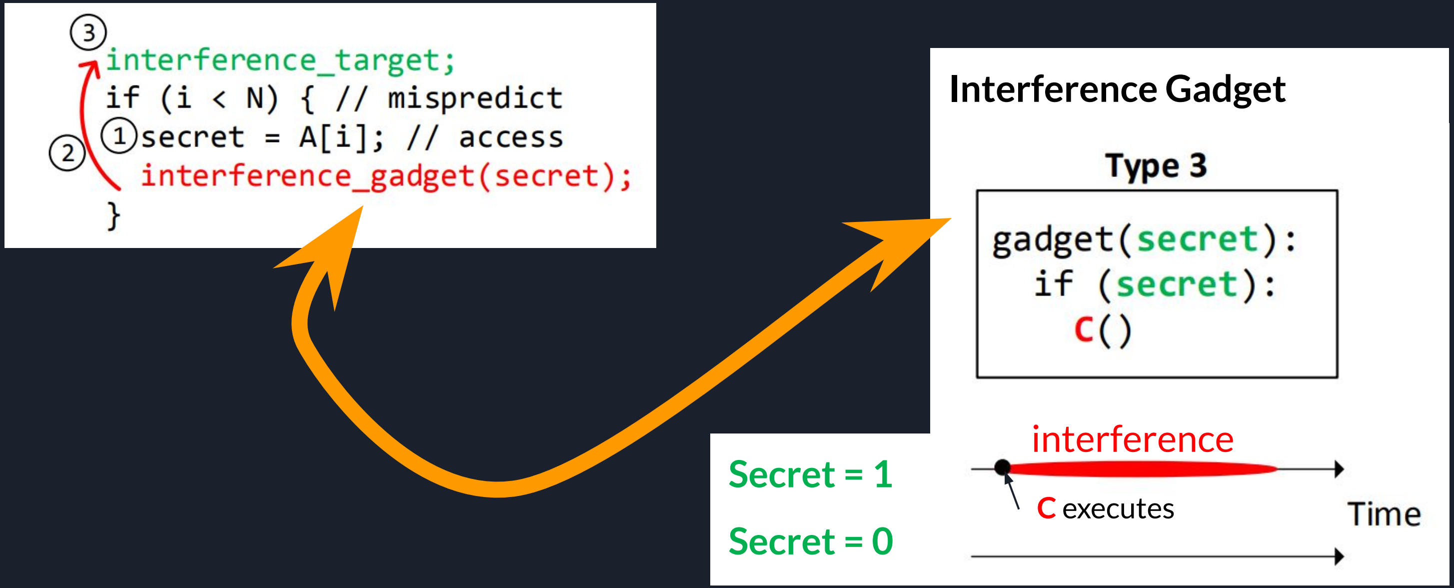
x = load(&S[secret*64])

f’(x)



Types of Gadgets

Attack Framework

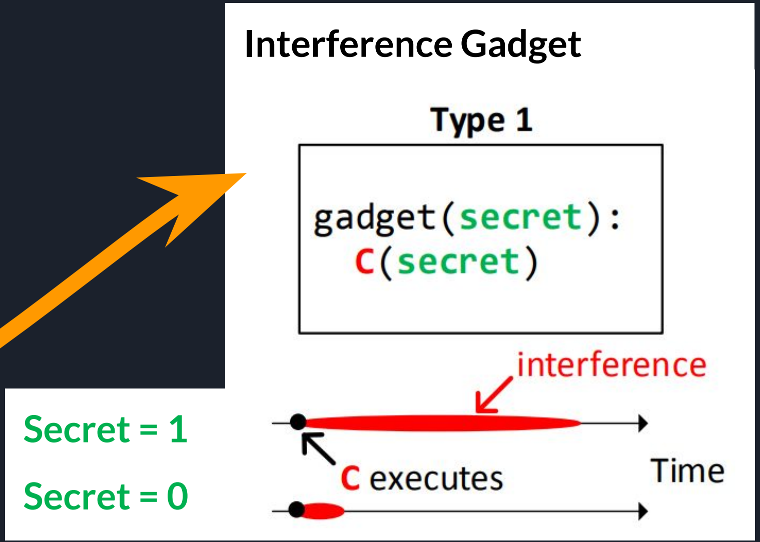


If secret = 1, interference_target gets delayed (delayed)

If secret = 0, interference_target executes immediately

Attack Framework

```
③
 interference_target;
 if (i < N) { // mispredict
②①secret = A[i]; // access
    interference_gadget(secret);
 }
```

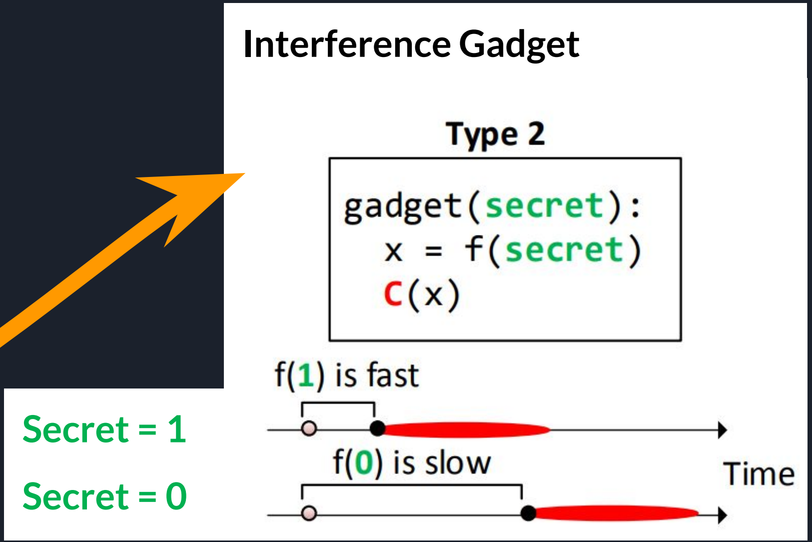


If secret = 1, interference_target gets delayed a **long** time

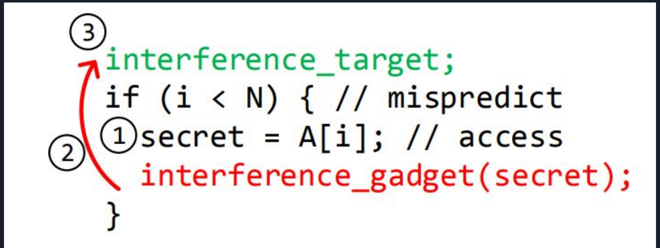
If secret = 0, interference_target gets delayed a **short** time

Attack Framework

```
③
  interference_target;
  if (i < N) { // mispredict
② ①secret = A[i]; // access
    interference_gadget(secret);
  }
```



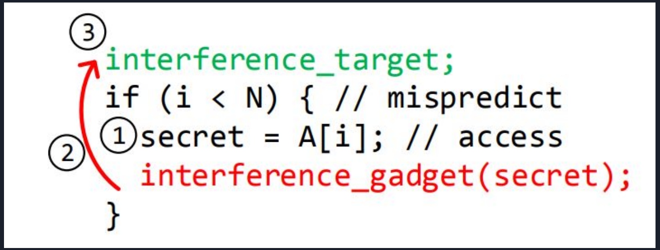
If secret = 1, interference_target gets delayed a **short** time
If secret = 0, interference_target gets delayed a **long** time



Depending on the secret, interference_target can get delayed

How is this useful?

Delays change instruction order



Interference_target:

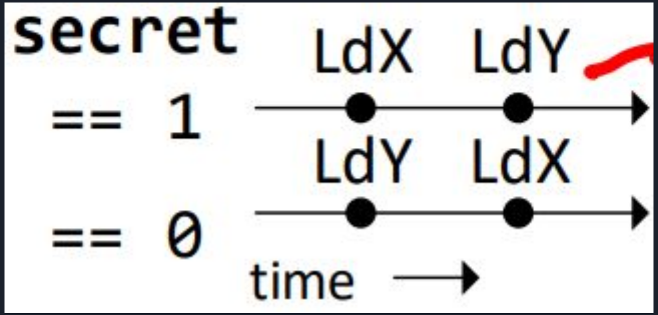
load(X)

load(Y)

Interference Gadget:

If secret = 1, does **NOT** delay load(X)

If secret = 0, **delays** load(X)



Conclusion:

If secret = 1, load(X) runs before load(Y)

If secret = 0, load(Y) runs before load(X)

Concrete example:

```
z = ...          // takes Z cycles
A = f(z)         // takes F cycles
y = load(A)
B = g(z)         // takes G > F cycles
v = load(B)
if (i < N):  // mispredict taken (miss on N)
    secret = load(&TargetArray[i])
    // Interference Gadget
    x = load(&S[secret * 64]) // secret=1->hit, secret=0->miss
    f'(x)
```

1. Prime cache so that secret=1 -> hit, secret = 0 -> miss
2. Observe:
   a. If secret = 1, **load(B)** happens before **load(A)**
   b. If secret = 0, **load(A)** happens before **load(B)**
3. From cache state, infer secret.
   a. Ensure, A and B are in the same cache set. Then start triggering evictions. LRU gets evicted first. If A = LRU → secret = 0. If B = LRU → secret = 1

Evaluation

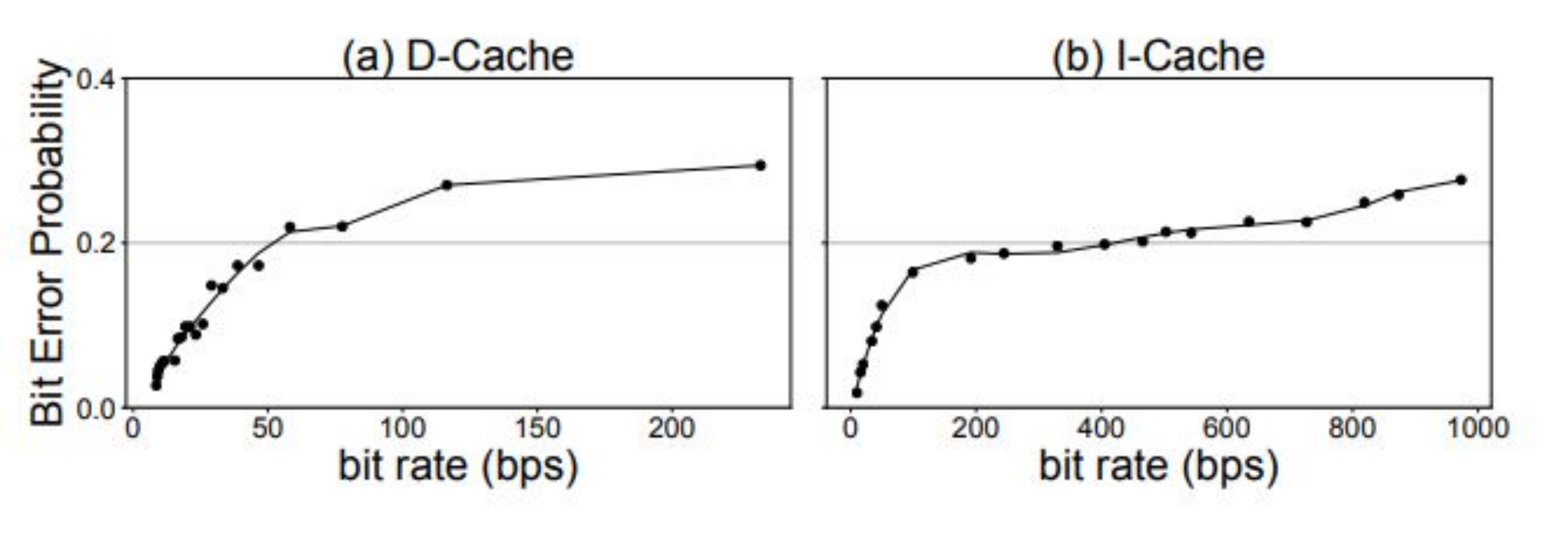
They evaluated on real machines despite invisible speculation schemes not being available

- By artificially making “invisible” loads secret dependent

Had to reverse the Kaby Lake D-Cache replacement policy

Worked on LLC, so cross core attack

Results - Sender and Receiver



Potential Impacts and Limitations

Gadgets have to be very specific. More useful in sandbox environments.

Cache protection systems don't exist yet, so this attack is mainly theoretical.

Brings up good points for future invisible speculation scheme designs

- Calls for the necessity of timing independent invisible speculation schemes (that don't change cache state)

Defense

Basic Defense Design:

- Execute speculative instructions but queue them up in the ROB and don't finish them until the oldest speculative instruction gets resolved

To fix Spectre, only do this for branches

Discussion Question

Is hardware / program vulnerability an inevitable byproduct of speculative fetches or is the overhead of performance that would come with an ideal invisible speculation scheme worth the security flaw?

Discussion Question

The paper evaluates its methods by sending secret zeros and ones after making many simplifying assumptions. Could this be used to actually leak meaningful data in the wild?