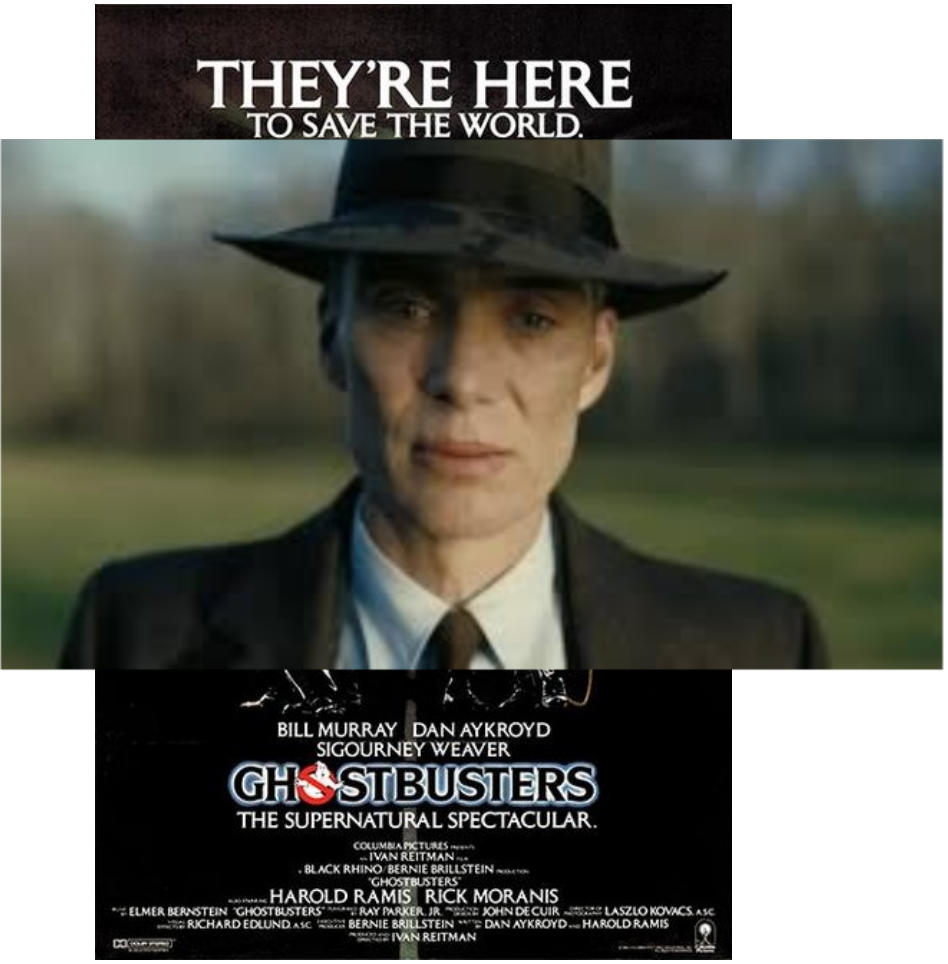
Language Based Security (LBT)

Secure compilation

Chiara Bodei, Gian-Luigi Ferrari

Lecture May, 20 2024



Becoming a Ghostbuster*

Chiara Bodei, Gian-Luigi Ferrari

We thank Matteo Busi for the possibility of using and modifying these nice slides

*actually, a Spectrebuster

Outline

Mechanisms, attacks and mitigations
More formally
Conclusions

Speculation

Speculation is based on predict

- Caches are **fast** when data is already there, **otherwise** we need to wait for the DRAM 😢
- **Idea:**
  - Upon branch on a value in memory (**uncached**), the CPU speculates on what's **probably right** to do and does that (while we load value)
  - When the value arrives, if it was the wrong thing: **rollback, but cache modifications are not restored**
  - **Transient executions are reverted at architectural level, but not at microarchitectural one**

The ones due to speculation

But cache modifications are not restored. PROBLEM

- For that, the CPU is equipped with a **branch predictor**
  - **Roughly:** should the CPU take the next branch?
  - The CPU **trains** its branch predictor by **observing** what happened before
  - Modern branch predictors are complex, but for our purposes **just one bit**!

Towards Spectre

Consider an attacker that just **observes** the *execution* of a program

- If it can learn some information from its observation, then there is a side channel!
- **Informally:** programs that resist against attackers able to measure time are said **constant-time programs**
  - **Actually:** wait till the end for a better (i.e., implementation-independent) definition

• Side channels have a time dimension

• A side channel is anything that on an implicit and unwanted way (leaks) information about a program

More concretely

- Assume an attacker can measure the execution time of a program
- Can it attack a program on a system with **caches? Cache-based covert channel**
- **Yes!** 😎 Roughly because caches introduce *timing variations,* based on the memory accesses
- Find a line from the cache shared between the attacker and the victim
  - **Flush/Evict** it from the cache
    - e.g., using clflush on x86 or by loading attacker's data ending up in the same cache line
  - Let the victim run for a while
  - Measure the time it takes to perform a memory read at the address corresponding to the evicted cache line (**Reload**)
    - If the victim accessed the shared line, the access will be fast (data was cached!)
    - Otherwise, the read will be slow
- Attacks are also possible without sharing memory

Exploit caching: Flash + Reload Attack

of course these considerations work when we have security and secrets

Actuallly, everyhing apart write including cache sharing

Suppose there is an encryption algorithm shared between the attacker and the victim [read-only hence considered safe]

```
1 if (secret == 1) {
2 x = x + 1;
3 }
4 else {
  …
```

Executed only when secret is 1

- The attacker performs a clflush (line 2) and lets the victim execute
- The attacker try to reload the line and measures the time
  - In case of cache miss: the victim **did not** access line 2
  - In case of cache hit: the victim **did** access the line 2

This observation says wether secret is 1 or not.

Assume that on cache line 2 we have X.

Exploit caching: Flash + Reload Attack

The FLUSH+RELOAD attack relies on a combination of 4 factors:

- data flow from sensitive data to memory access patterns
- memory sharing between the attacker and the victim
- accurate, high-resolution time measurements and
- the unfettered use of the clflush instruction

Preventing any of these blocks the attack

Exploit caching: Evict + Reload Attack

Evict+Reload forces contention on the cache set that stores the line, causing the processor to discard the contents of that cache line by loading something else instead of flushing it

The attacker evicts

Victim accesses/does not access

Attacker reloads

- Fast access time -> victim accessed 
- Slow access time -> victim did not access 

Now about this line:

> "Evict+Reload forces contention on the cache set that stores the line, causing the processor to discard the contents of that cache line by loading something else instead of flushing it"

Let's rephrase it casually:

- It's saying the attacker **doesn't need to explicitly flush** (clear) the cache like in other attacks (e.g., Flush+Reload).
- Instead, they "**cause contention**" — meaning they fill the cache set with other stuff, pushing the victim's data out naturally. That's the "evict" part.
- So the cache line gets overwritten or "discarded" **because something else takes its place**, not because the attacker directly told the system to flush it.

Exploit caching: Prime + Probe Attack

No shared memory between the attacker and the victim

- The victim has access to a variable a and
- the attacker has access to a variable c s.t. a and c map to the same cache line
- The attacker evicts the address &a from the cache by accessing the address &c and lets the victim execute
- If the attacker try to access &c:
  - In case of cache miss: the victim **did** access &a 
  - In case of cache hit: the victim **did not** access &a 

depends on cache models: 2 blocks can go to the same line.

Indirect way to exploit shared

```
if (secret == 1) {
maccess(&a);
}
else {
  ...
```

access the address a