Incentivizing Side-Channel Freedom with Leakage-Aware Computation

Gongqi Huang
Princeton University
gongqih@princeton.edu

Jingyuan Chen
Princeton University
leocjy@princeton.edu

Amit Levy
Princeton University
aalevy@princeton.edu

Abstract

Microarchitectural side-channels have long been recognized as a serious security threat. While the systems research community has proposed new architecture and system design to close such channels, infrastructure providers continue to deploy systems that rely on *ad hoc* mitigations. These providers have little incentive to guarantee side-channel freedom, as doing so requires additional effort, and victims have no evidence to hold providers accountable. To address this, instead of searching through the system design space for an ultimate solution, we make microarchitectural side-channel leakage tangible to victims through *leakage-aware computation*, thereby incentivizing providers to guarantee side-channel freedom. As a starting point, we argue that making computations leakage-aware is both feasible and practical for *cache* side-channel leakage.

1 Introduction

Microarchitectural side-channels have long been recognized as a serious security threat, as they subvert the fundamental security guarantee of computer systems. Incidents have demonstrated the practicality of exploiting them on real, commodity hardware [4–6]. While the systems research community has proposed new architecture and system designs to close such channels, infrastructure providers continue to deploy systems that rely on *ad hoc* mitigations[1] [2, 3, 7].

These providers have little incentive to guarantee side-channel freedom: deploying leakage-free systems requires additional effort, and their effectiveness has yet to be battle-tested. More importantly, side-channel leakages often do not affect functional correctness and are thus difficult to notice. As a result, even when leakage occurs, victims have no evidence to hold providers accountable. Instead of searching through system design space for an ultimate solution, we make microarchitectural side-channel leakage tangible to victims through *leakage-aware computation*, thereby incentivizing providers to guarantee side-channel freedom. To begin, we argue that making computations leakage-aware is both feasible and practical for *cache* side-channel leakage.

The insight behind our methodology is that both attacker and victim interact with a shared resource via a *common* interface. A successful attack infers secrets by *observing* changes in the state. If *any* observation always causes changes to the state, there *must* be a way for the victim to detect the presence of an attack.

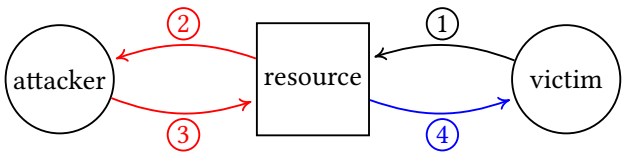

Figure 1. Symmetric Information Flow.

This condition is not just hypothetical. In practice, we have observed that it holds across many real-world resource interfaces (e.g., cache, interconnects). This behavior is illustrated in Figure 1. When the attacker learns some secret through ① → ②, they also create a flow ③ that captures this attack. The victim can later detect this attack through ③ → ④, and is thus *leakage-aware*.

Based on our insight, we put forth the "Symmetry Conjecture of Side Channels": The power of the attacker and that of the victim are symmetric with respect to a common interface over a shared resource. We specify this conjecture in the following statement:

$$\forall I_e, (\exists h, E_h(I_e)) \rightarrow (\exists h, E_h(E_h(I_e)))$$

In the conjecture, I_e represents a piece of information about an event e (e.g.,"e accessed address X") that the attacker is interested in; h stands for a valid history of events in the system that contains e; $E_h(I_e)$ is a proposition that states "I_e is exposed in history h", meaning that I_e can be learned by the attacker through the interface in h. The full conjecture states that if it is possible for I_e to be exposed in some valid history, then it is also possible for the information "I_e is exposed" to be exposed in some valid history. Hence, if the conjecture holds, then for any information that the attacker can learn through the interface, the victim has a way to detect such a leakage through the same interface.

While eliminating microarchitectural side-channels in practical systems may not be feasible in the near future, our methodology can at least install a "check engine light" in applications, providing users with a sense of security. We hope users will make judicious decisions with this information, rather than "drive to the shoreline with the check engine light on[2]."

[1] We can catch a glimpse of this iceberg by running `lscpu`.

[2] Lyrics from *Cinco de Mayo Shit Show* by *Marietta*.

```
1 if secret {
2   doFoo;
3 } else {
4   doBoo;
5 }
```

Listing 1. Leaky Program P.

2 With Check Engine Light

Our goal is to have a program enhancer $\mathcal{P} \rightarrow \mathcal{P}^\star$ that converts a given program P to its functionally-equivalent version $P^\star$. $P^\star$ is capable of detecting and alerting to any of its cache side-channel leakage that occurs, making it *leakage-aware*. The construction of $P^\star$ involves three phases: 1) Preload phase: all secret-dependent memory accesses are preloaded into the cache; 2) Execution phase: P is executed, and the latency of each secret-dependent memory access is recorded; 3) Check phase: the observed latencies are checked against expected values, and any mismatch implies a cache side-channel leakage. The detection is possible because the preload phase ensures that, in the absence of leakage, the observable cache state is known. In this section, we illustrate this process with an example and provide a preliminary argument for its correctness. Finally, we briefly discuss the challenges and solutions to making this approach practical.

2.1 Leaky Program P

Listing 1 shows a leaky program P — a program that contains a one-bit-secret-dependent branch. Each possible path executes a single instruction, with no additional memory access other than its own instruction fetch. During the execution of P, we assume a direct mapped cache with a line size equal to that of one instruction. Furthermore, the attacker has full control over a common cache interface (i.e., memory access and cache line eviction) at any time and has full knowledge of P's text layout.

In this case, P is leaky because its cache footprint depends on the secret. The attacker can infer the secret by checking whether, for example, instruction `doFoo` is in the cache.

2.2 Strawman Leakage-Aware Program $P^\star$

We present our strawman $P^\star$ in Listing 2. `now!()` is a macro that inserts a timestamp instruction[3] (e.g., `RDTSC` in x86, `RDCYCLE` in RISC-V), while `time!()` measures execution time by inserting two timestamp instructions.

2.2.1 Informal Argument. We now provide an informal argument that $P^\star$ is indeed leakage-aware. To begin, we assume that $P^\star$ does not evict its own cache lines, and that

```
1  let _  = (*la, *lb, *lc, *ld);
2  let t0 = now!();
3  let t1 = if secret {
4    la: doFoo; lb: now!()
5  } else {
6    lc: doBoo; ld: now!()
7  };
8  if t1 - t0 >= THRES
9  || time!({ let _ = *la; }) >= THRES
10 || ... // Omit checks for lb, lc, ld
11    { panic!("Leaked!"); }
```

Listing 2. Leakage-Aware Program $P^\star$.

there exists a latency threshold, `THRES`, which reliably distinguishes between a cache hit and miss.

To infer the secret, the attacker must *evict* at least one cache line of interest (i.e., cache lines touched by the branch) and *observe* changes in a cache line. Suppose both the eviction and observation occur before the branch (line 3), it is impossible for the attacker to succeed, as no cache lines are secret-dependent yet. If the eviction occurs before the branch and the observation occurs right after, the checks (line 8-11) will capture the eviction. Since all cache lines of interest have been preloaded into the cache (line 1), their eviction must result in a cache miss that is detectable by the victim. Finally, if both actions happen after the branch, the attacker cannot infer the secret because all accesses are not secret-dependent. Therefore, we have shown $P^\star$ is leakage-aware with respect to the cache side-channel.

2.3 Challenges and Solutions

The main challenge to make the solution practical is that it requires every secret-dependent memory access to be instrumented, which introduces significant performance overhead[4]. Several workarounds are possible. First, we can reduce the number of timestamp instructions while still preserving detection soundness. Alternatively, we can reduce the timestamp instruction's latency by using a separate software thread (where a timestamp instruction is a cached memory write) or implementing a faster timestamp instruction. Finally, non-timing-critical checks can be delayed until the end, allowing earlier return of P's result.

3 Conclusion

Achieving side-channel freedom isn't necessarily a dream. With the check engine light on, maybe all we lack is the will?

[3] We assume a timestamp instruction that has the semantics of returning the current timestamp after it is fetched from memory.

[4] For example, a single `RDTSC` requires 46 μops in AMD Zen 4 [1].

References

[1] Agner Fog. 2022. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. https://www.agner.org/optimize/instruction_tables.pdf Retrieved at 2025-03-24.

[2] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time Protection: The Missing OS Abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 1, 17 pages. https://doi.org/10.1145/3302424.3303976

[3] Gongqi Huang, Leon Schuermann, and Amit Levy. 2024. Bridge: A Leak-Free Hardware-Software Architecture for Parallel Embedded Systems. In *Proceedings of the 2nd Workshop on Kernel Isolation, Safety and Verification* (Austin, TX, USA) *(KISV '24)*. Association for Computing Machinery, New York, NY, USA, 16–22. https://doi.org/10.1145/3698576.3698765

[4] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[5] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[6] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 679–697. https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen

[7] Ziqiao Zhou, Yizhou Shan, Weidong Cui, Xinyang Ge, Marcus Peinado, and Andrew Baumann. 2023. Core slicing: closing the gap between leaky confidential VMs and bare-metal cloud. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 247–267. https://www.usenix.org/conference/osdi23/presentation/zhou-ziqiao