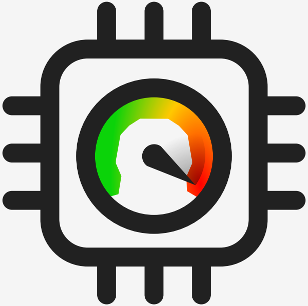
CISPA
HELMHOLTZ CENTER FOR
INFORMATION SECURITY

PreFence: A Fine-Grained and Scheduling-Aware Defense Against Prefetching-Based Attacks

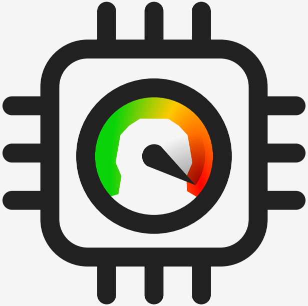
Till Schlüter, Nils Ole Tippenhauer

IEEE European Symposium on Security and Privacy (EuroS&P) 2025

Motivation

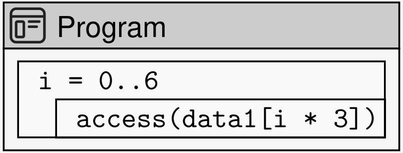


Motivation



Hardware Prefetching

Example: Stride Prefetching



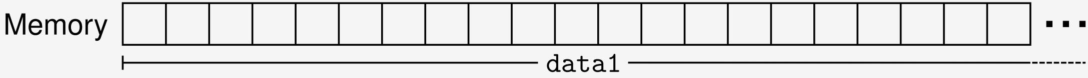
Program

```
i = 0..6
    access(data1[i * 3])
```

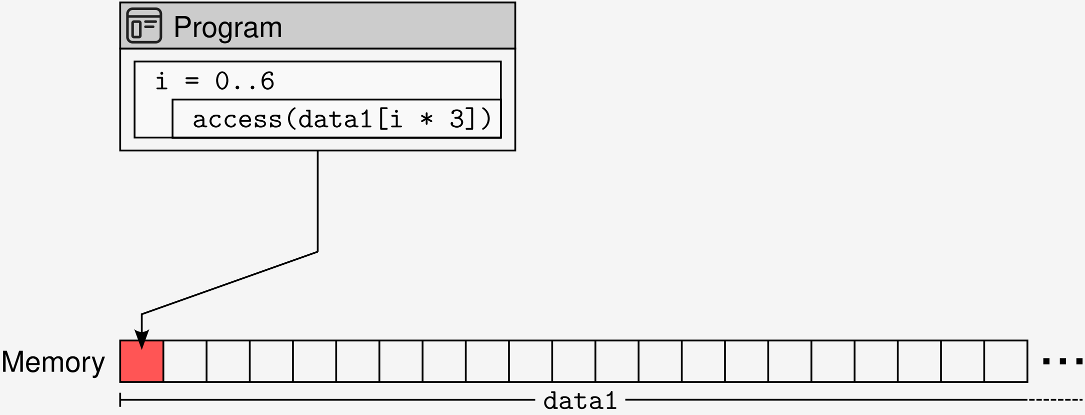
Example: Stride Prefetching

Program

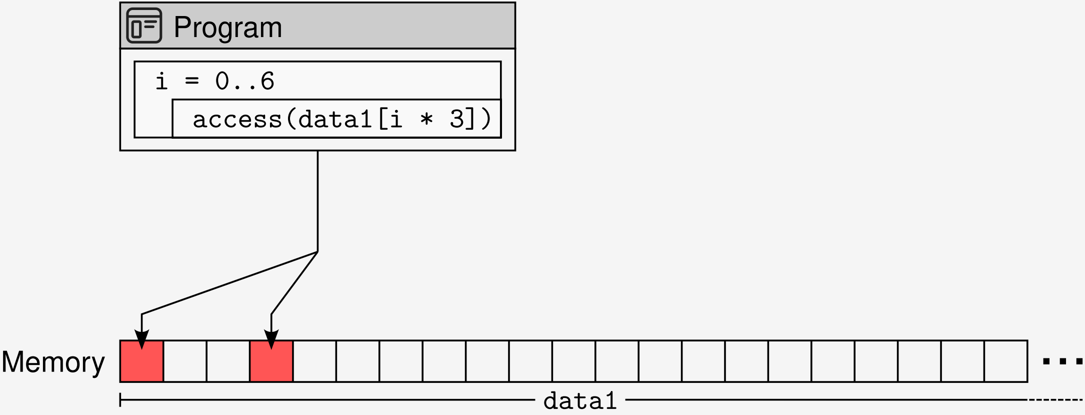
```
i = 0..6
    access(data1[i * 3])
```



Example: Stride Prefetching



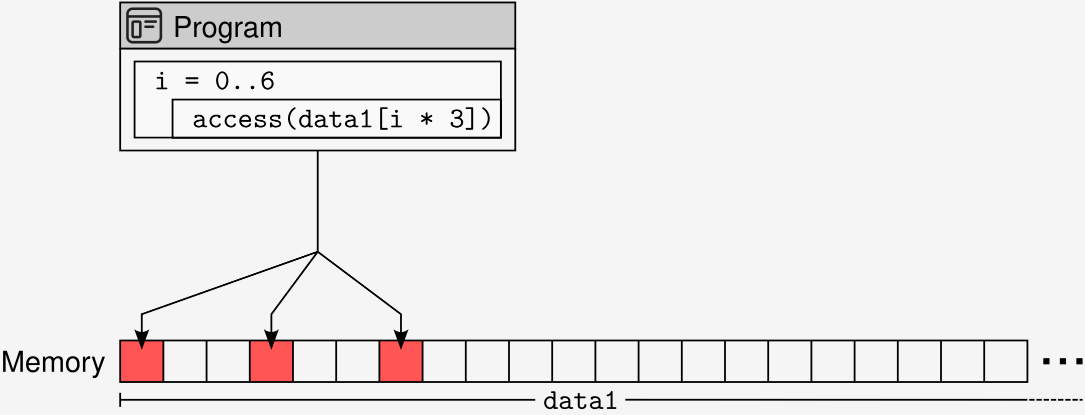
Example: Stride Prefetching



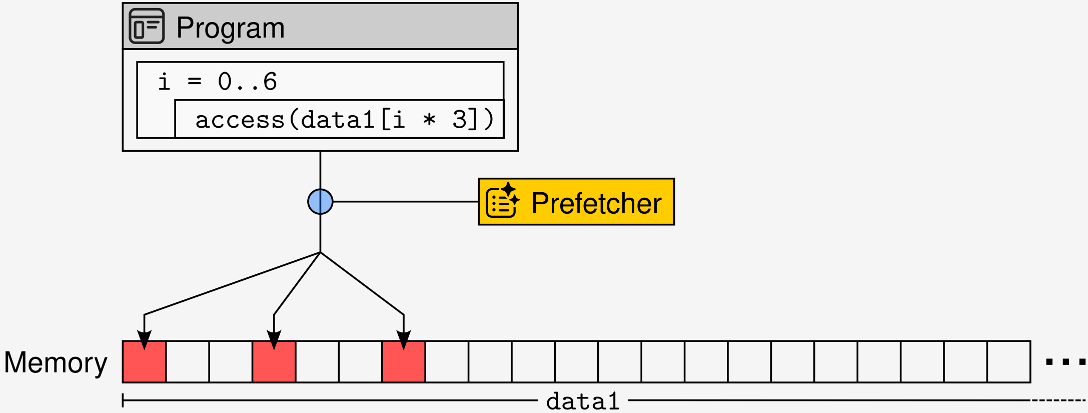
Program

```
i = 0..6
    access(data1[i * 3])
```

Example: Stride Prefetching

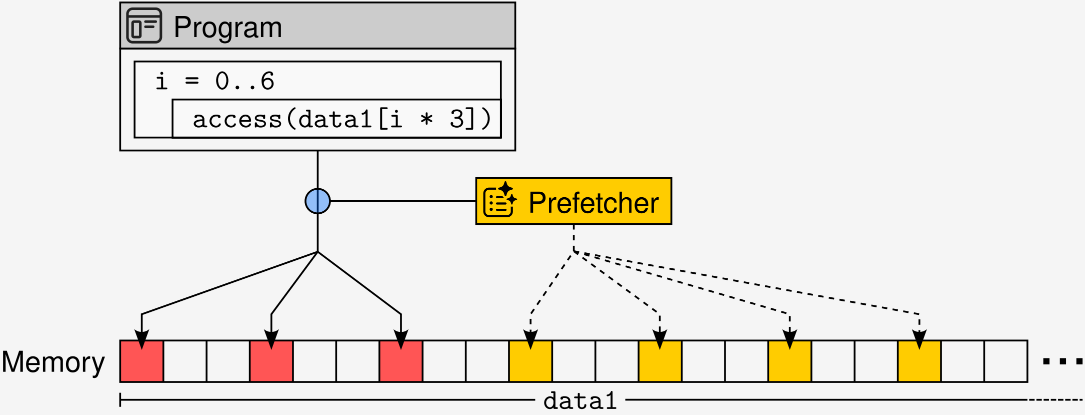


Example: Stride Prefetching



```
i = 0..6
    access(data1[i * 3])
```

Example: Stride Prefetching



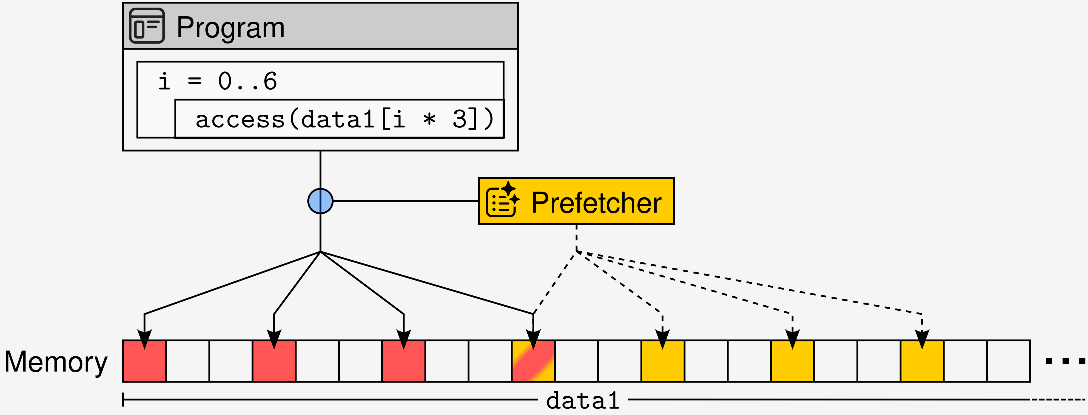
Program

```
i = 0..6
    access(data1[i * 3])
```

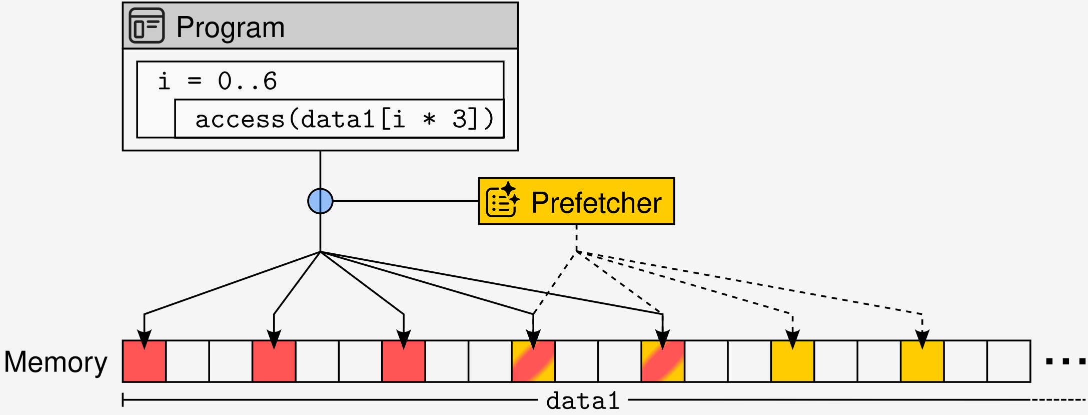
Prefetcher

Memory

Example: Stride Prefetching

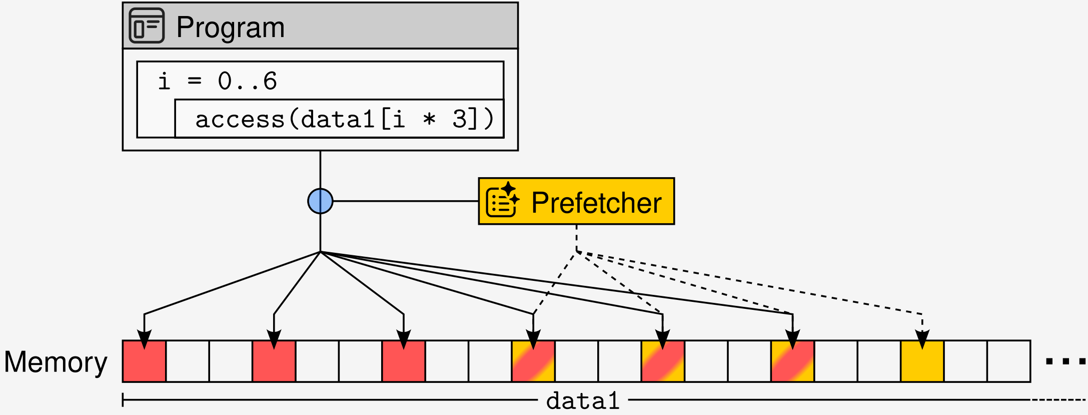


Example: Stride Prefetching

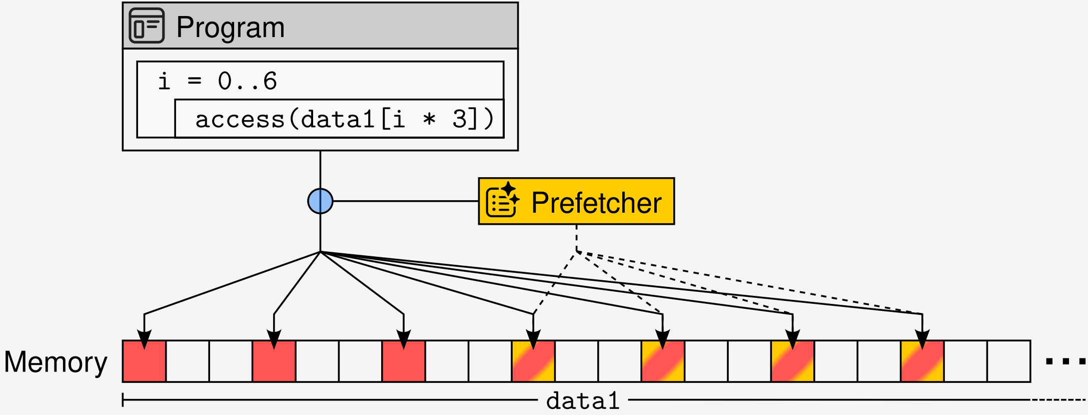


```
i = 0..6
    access(data1[i * 3])
```

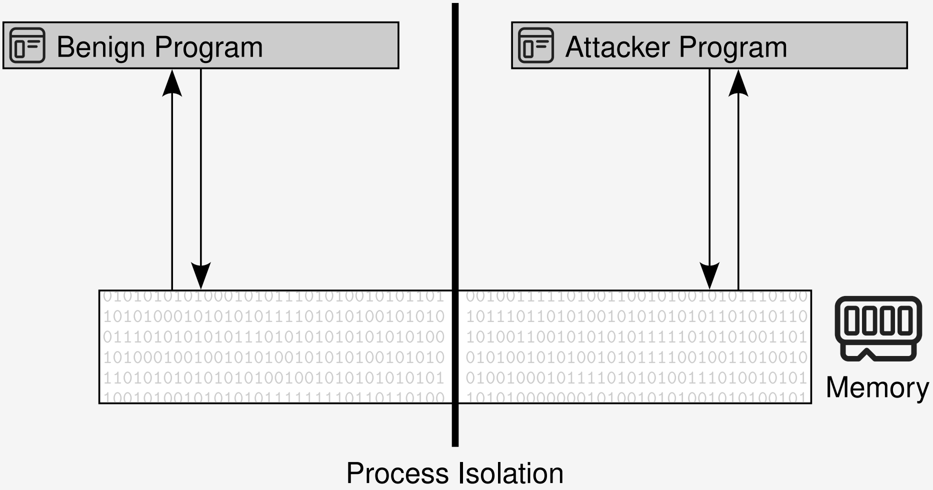
Example: Stride Prefetching



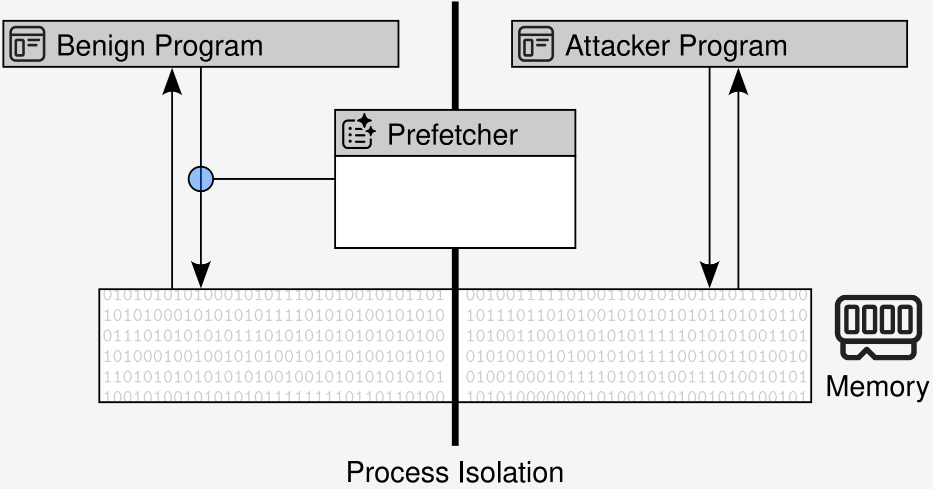
Example: Stride Prefetching



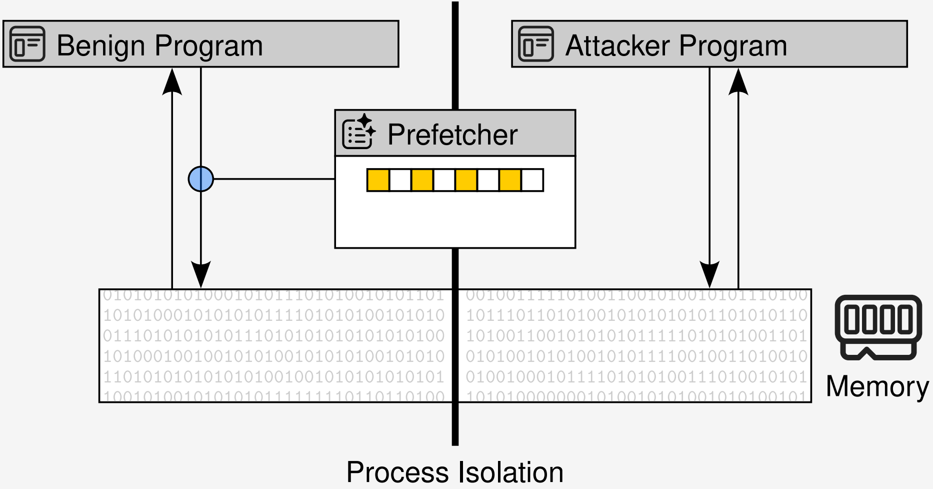
Example: Prefetch Attack



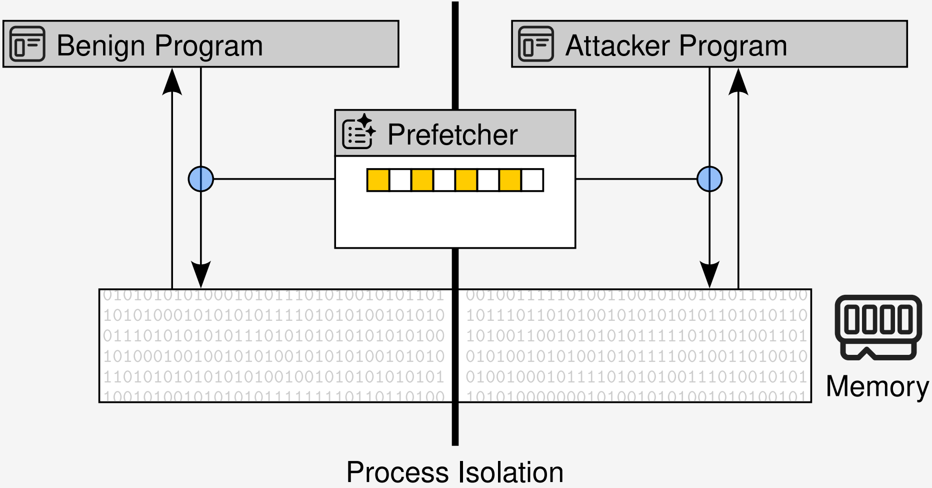
Example: Prefetch Attack



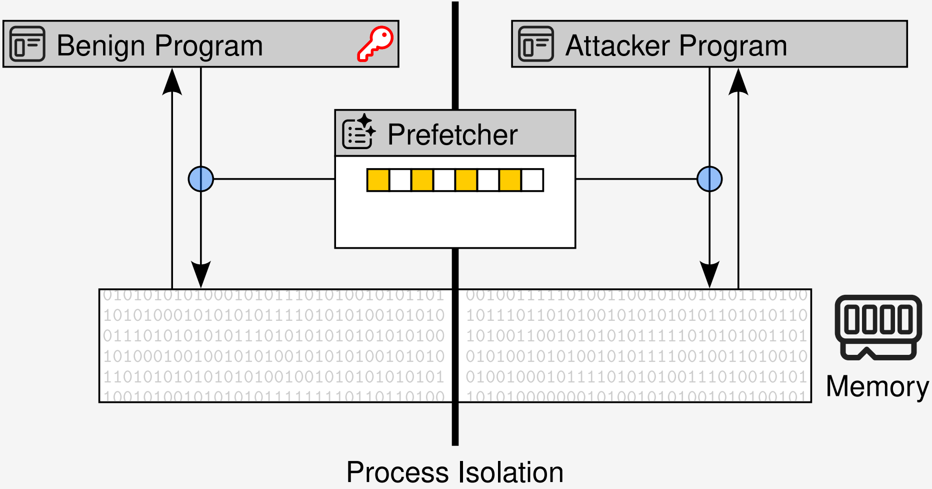
Example: Prefetch Attack



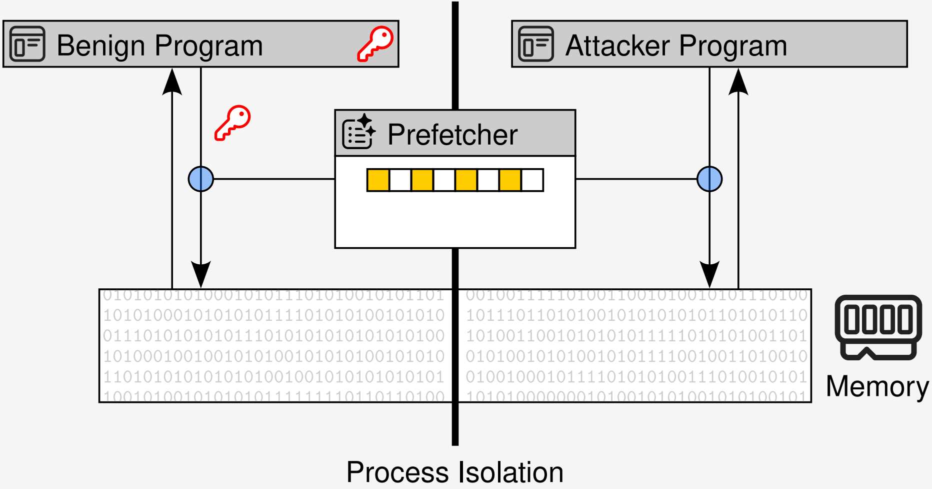
Example: Prefetch Attack



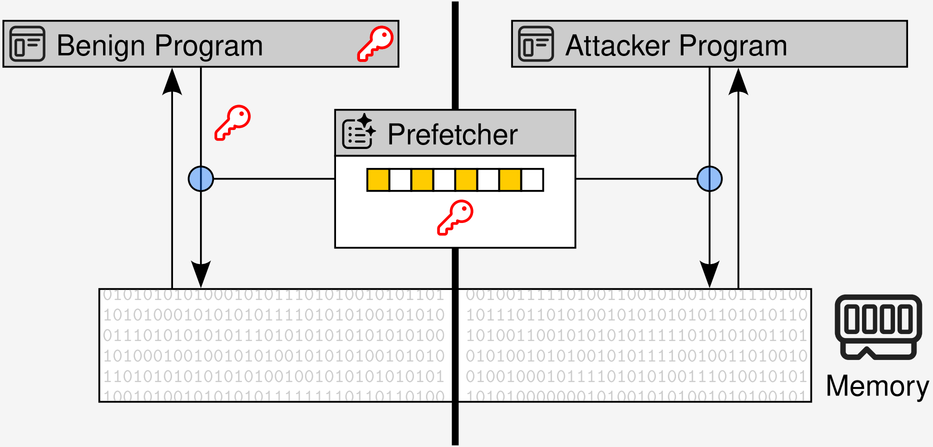
Example: Prefetch Attack



Example: Prefetch Attack

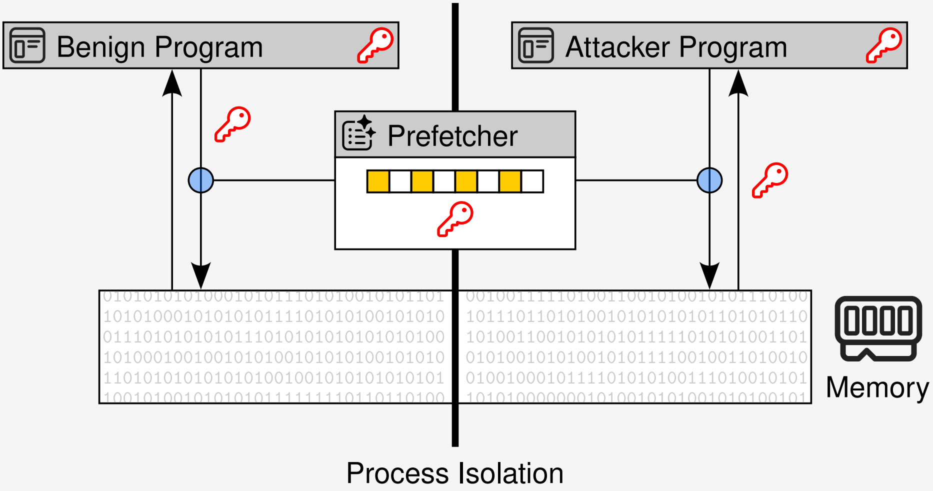


Example: Prefetch Attack



Process Isolation

Example: Prefetch Attack



Defenses So Far

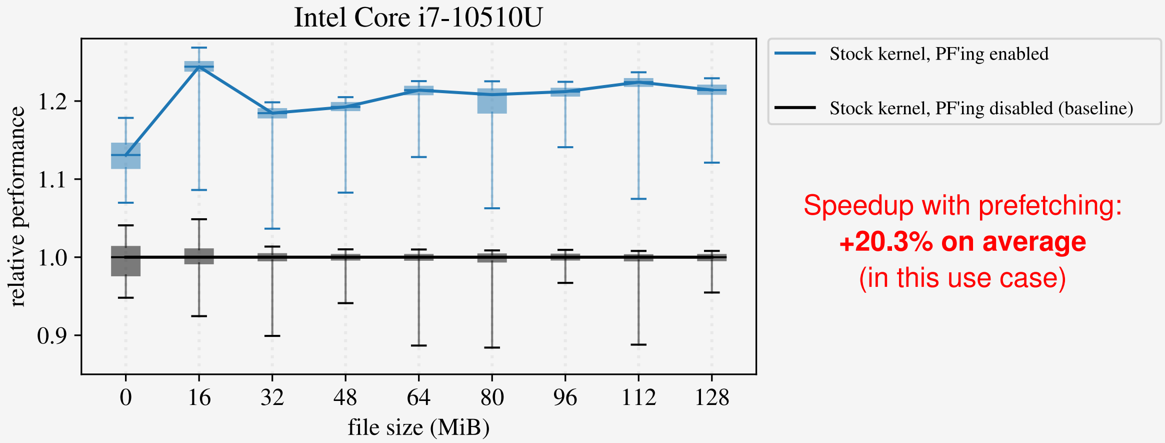
Targeted defenses

Defenses So Far

Targeted defenses

Disable prefetching

Disabling Prefetching Is Expensive



Speedup with prefetching:
+20.3% on average
(in this use case)

Design Goals

Can we find a defense that...

Design Goals

Can we find a defense that...

...prevents prefetch attacks

Design Goals

Can we find a defense that...

...prevents prefetch attacks

...has minimal runtime overhead

Design Goals

Can we find a defense that...

...prevents prefetch attacks

...has minimal runtime overhead

...is easy to use
for developers and end users

Design Goals

Can we find a defense that...

...prevents prefetch attacks

...has minimal runtime overhead

...is easy to use
for developers and end users

...is compatible with
Simultaneous Multithreading (SMT)

Prefetching-Based Side-Channel Attacks in Prior Work

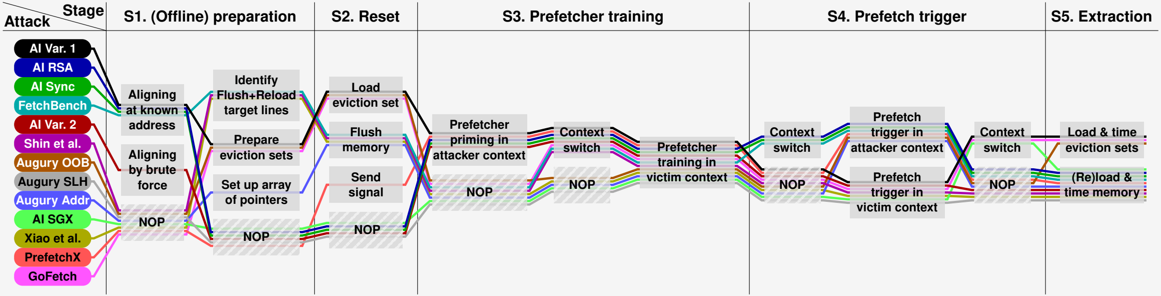
We consider 13 attacks from 7 papers:

#	Attack	Prefetcher
1	Shin et al.	Intel IP stride
2	Augury OOB	Apple DMP
3	Augury SLH	Apple DMP
4	Augury Addr.	Apple DMP
5	AfterImage Var. 1	Intel IP stride
6	AfterImage Var. 2	Intel IP stride
7	AfterImage SGX	Intel IP stride
8	AfterImage RSA	Intel IP stride
9	AfterImage Sync	Intel IP stride
10	Xiao et al.	Intel IP stride
11	FetchBench AES	ARM SMS
12	PrefetchX	Intel XPT
13	GoFetch	Apple DMP

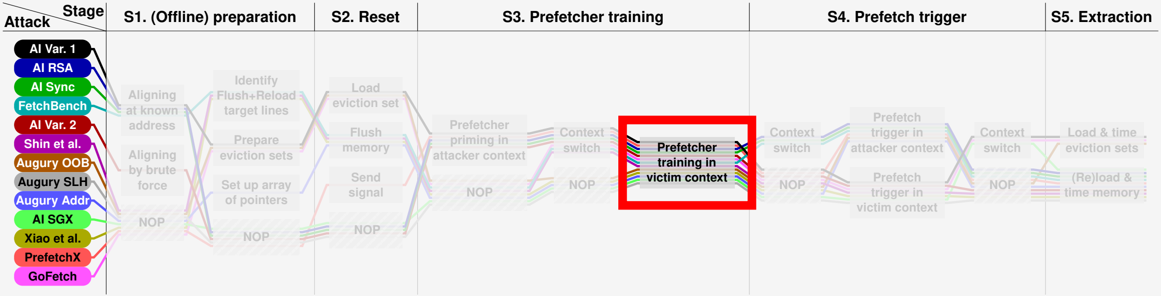
Attack Systematization

Stage Attack
AI Var. 1
AI RSA
AI Sync
FetchBench
AI Var. 2
Shin et al.
Augury OOB
Augury SLH
Augury Addr
AI SGX
Xiao et al.
PrefetchX
GoFetch

Attack Systematization



Attack Systematization



Attack / Stage

S1. (Offline) preparation

S2. Reset

S3. Prefetcher training

S4. Prefetch trigger

S5. Extraction

AI Var. 1
AI RSA
AI Sync
FetchBench
AI Var. 2
Shin et al.
Augury OOB
Augury SLH
Augury Addr
AI SGX
Xiao et al.
PrefetchX
GoFetch

Aligning at known address
Aligning by brute force
NOP
Identify Flush+Reload target lines
Prepare eviction sets
Set up array of pointers
NOP
Load eviction set
Flush memory
Send signal
NOP
Prefetcher priming in attacker context
NOP
Context switch
NOP
Prefetcher training in victim context
Context switch
NOP
Prefetch trigger in attacker context
Prefetch trigger in victim context
Context switch
NOP
Load & time eviction sets
(Re)load & time memory

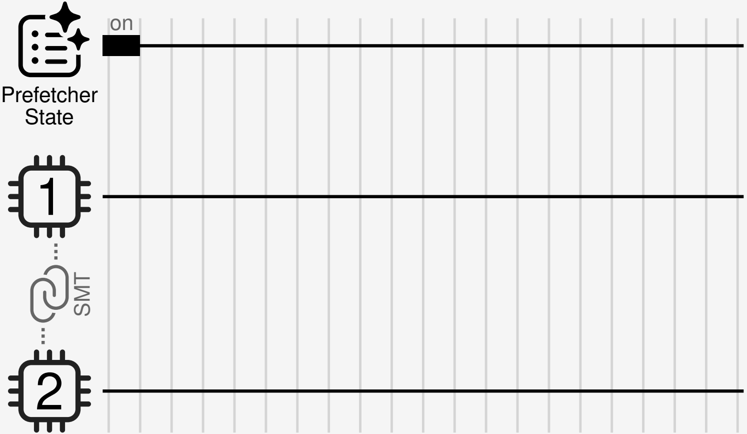
Finding: Victim process trains the prefetcher

Design Idea

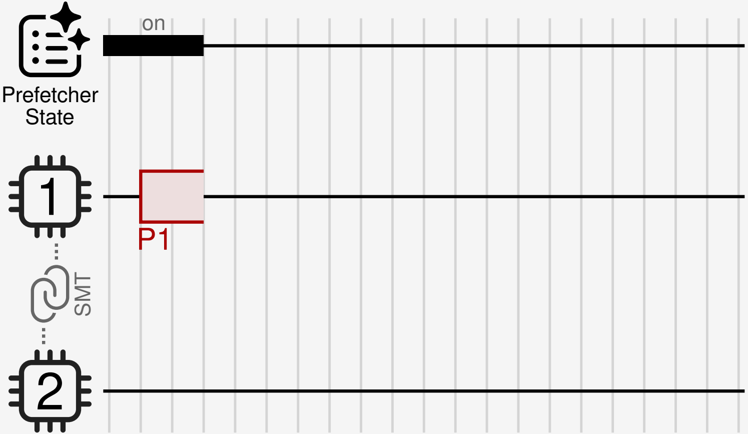
Disable prefetching
permanently

Disable prefetching
temporarily

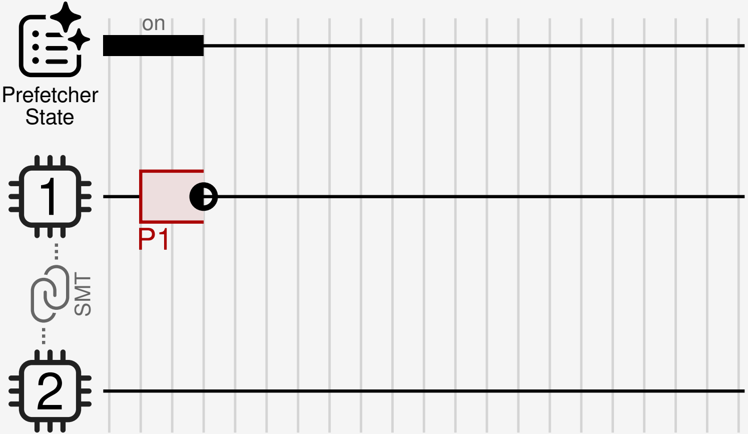
PreFence Design: Scheduling-Aware, Temporary Prefetcher Deactivation



PreFence Design: Scheduling-Aware, Temporary Prefetcher Deactivation

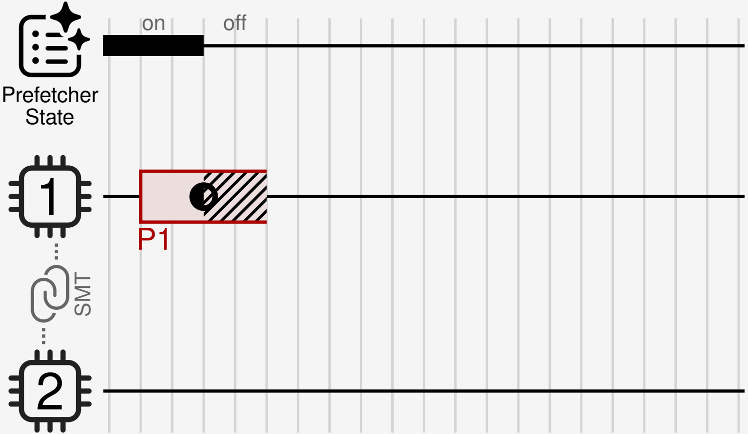


PreFence Design: Scheduling-Aware, Temporary Prefetcher Deactivation



◐ Signal: disable prefetching

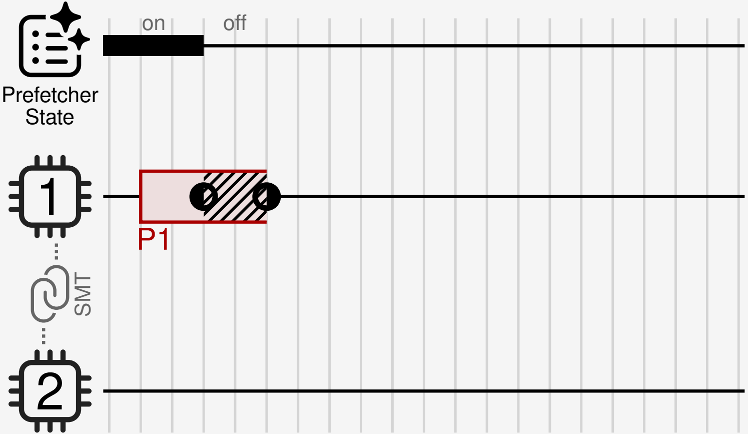
PreFence Design: Scheduling-Aware, Temporary Prefetcher Deactivation



◐ Signal: disable prefetching

▨ Security-critical computation

PreFence Design: Scheduling-Aware, Temporary Prefetcher Deactivation

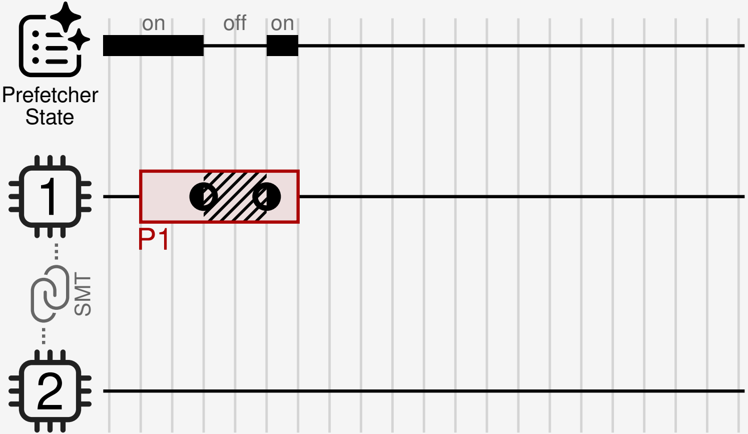


◐ Signal: disable prefetching

▨ Security-critical computation

◑ Signal: enable prefetching

PreFence Design: Scheduling-Aware, Temporary Prefetcher Deactivation

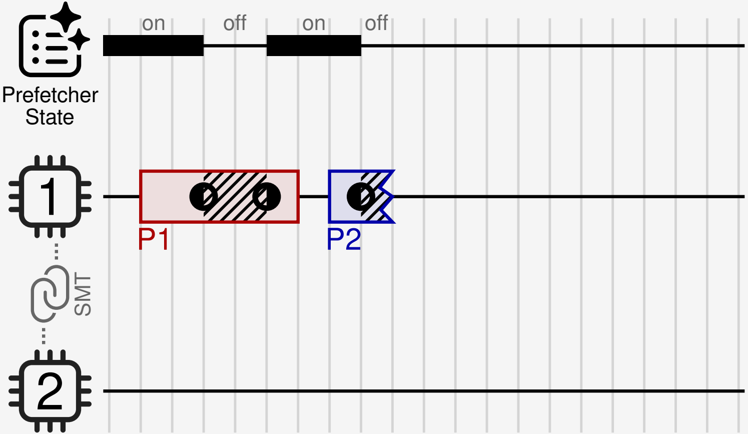


◐ Signal: disable prefetching

▨ Security-critical computation

◑ Signal: enable prefetching

PreFence Design: Scheduling-Aware, Temporary Prefetcher Deactivation

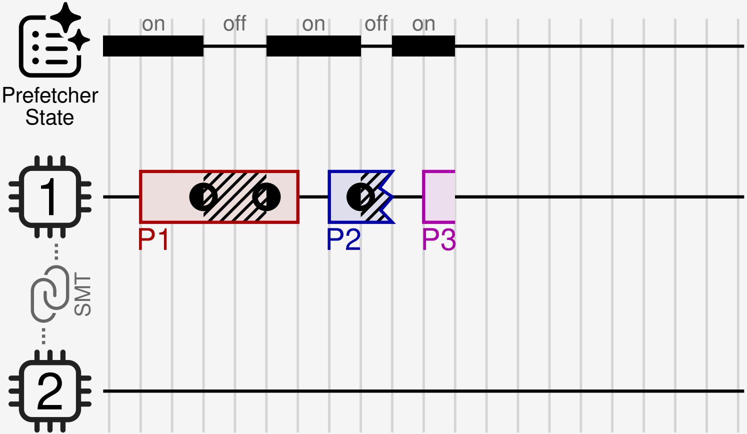


◐ Signal: disable prefetching

▨ Security-critical computation

◑ Signal: enable prefetching

PreFence Design: Scheduling-Aware, Temporary Prefetcher Deactivation

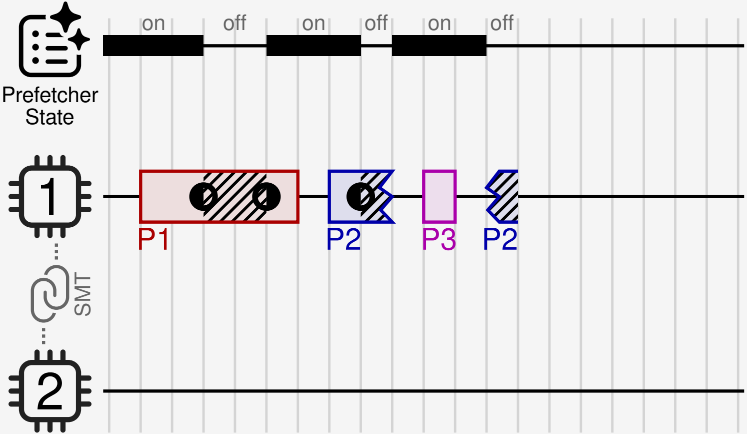


◐ Signal: disable prefetching

▨ Security-critical computation

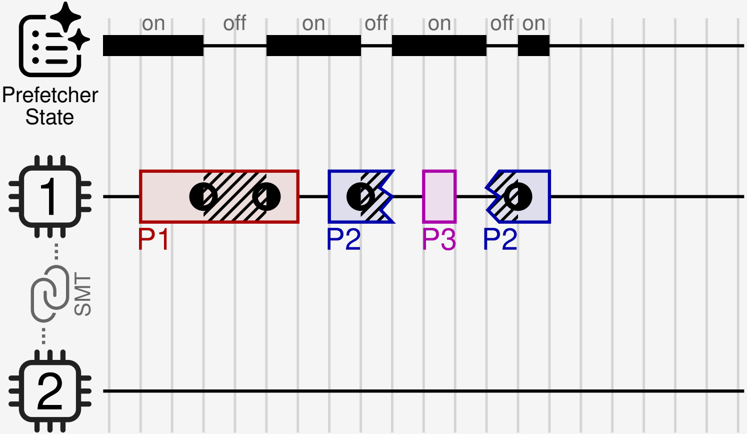
◑ Signal: enable prefetching

PreFence Design: Scheduling-Aware, Temporary Prefetcher Deactivation



- ◐ Signal: disable prefetching
- ▨ Security-critical computation
- ◑ Signal: enable prefetching

PreFence Design: Scheduling-Aware, Temporary Prefetcher Deactivation

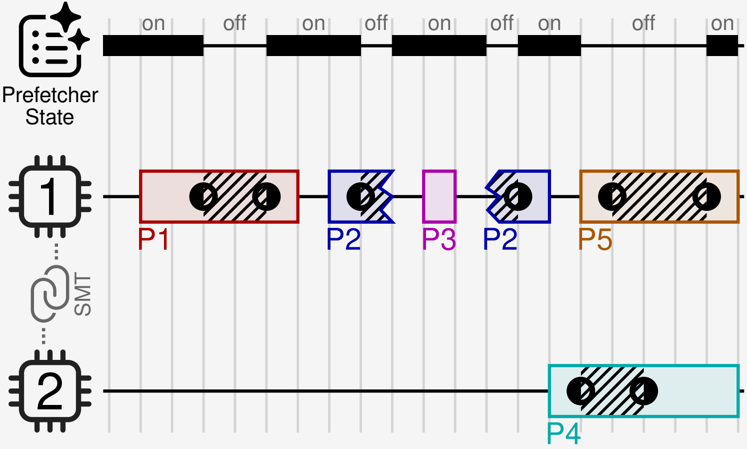


◐ Signal: disable prefetching

▨ Security-critical computation

◑ Signal: enable prefetching

PreFence Design: Scheduling-Aware, Temporary Prefetcher Deactivation



◐ Signal: disable prefetching

▨ Security-critical computation

◑ Signal: enable prefetching

Evaluation Targets

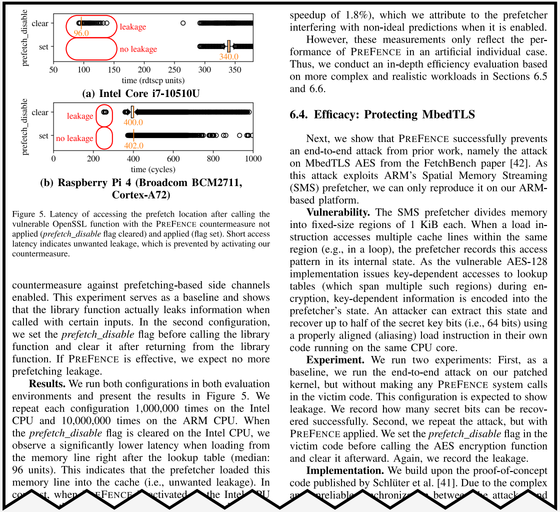
X86

Intel i7-10510U
(Comet Lake)

ARM

Arm Cortex-A72
(Broadcom BCM2711)

Efficacy: Prevents Prior-Work Attacks



(a) Intel Core i7-10510U

(b) Raspberry Pi 4 (Broadcom BCM2711, Cortex-A72)

Figure 5. Latency of accessing the prefetch location after calling the vulnerable OpenSSL function with the PREFENCE countermeasure not applied (*prefetch_disable* flag cleared) and applied (flag set). Short access latency indicates unwanted leakage, which is prevented by activating our countermeasure.

countermeasure against prefetching-based side channels enabled. This experiment serves as a baseline and shows that the library function actually leaks information when called with certain inputs. In the second configuration, we set the *prefetch_disable* flag before calling the library function and clear it after returning from the library function. If PREFENCE is effective, we expect no more prefetching leakage.

Results. We run both configurations in both evaluation environments and present the results in Figure 5. We repeat each configuration 1,000,000 times on the Intel CPU and 10,000,000 times on the ARM CPU. When the *prefetch_disable* flag is cleared on the Intel CPU, we observe a significantly lower latency when loading from the memory line right after the lookup table (median: 96 units). This indicates that the prefetcher loaded this memory line into the cache (i.e., unwanted leakage). In contrast, when PREFENCE is activated on the Intel CPU

speedup of 1.8%), which we attribute to the prefetcher interfering with non-ideal predictions when it is enabled.

However, these measurements only reflect the performance of PREFENCE in an artificial individual case. Thus, we conduct an in-depth efficiency evaluation based on more complex and realistic workloads in Sections 6.5 and 6.6.

6.4. Efficacy: Protecting MbedTLS

Next, we show that PREFENCE successfully prevents an end-to-end attack from prior work, namely the attack on MbedTLS AES from the FetchBench paper [42]. As this attack exploits ARM's Spatial Memory Streaming (SMS) prefetcher, we can only reproduce it on our ARM-based platform.

Vulnerability. The SMS prefetcher divides memory into fixed-size regions of 1 KiB each. When a load instruction accesses multiple cache lines within the same region (e.g., in a loop), the prefetcher records this access pattern in its internal state. As the vulnerable AES-128 implementation issues key-dependent accesses to lookup tables (which span multiple such regions) during encryption, key-dependent information is encoded into the prefetcher's state. An attacker can extract this state and recover up to half of the secret key bits (i.e., 64 bits) using a properly aligned (aliasing) load instruction in their own code running on the same CPU core.

Experiment. We run two experiments: First, as a baseline, we run the end-to-end attack on our patched kernel, but without making any PREFENCE system calls in the victim code. This configuration is expected to show leakage. We record how many secret bits can be recovered successfully. Second, we repeat the attack, but with PREFENCE applied. We set the *prefetch_disable* flag in the victim code before calling the AES encryption function and clear it afterward. Again, we record the leakage.

Implementation. We build upon the proof-of-concept code published by Schlüter et al. [41]. Due to the complex and unreliable synchronization between the attacker and

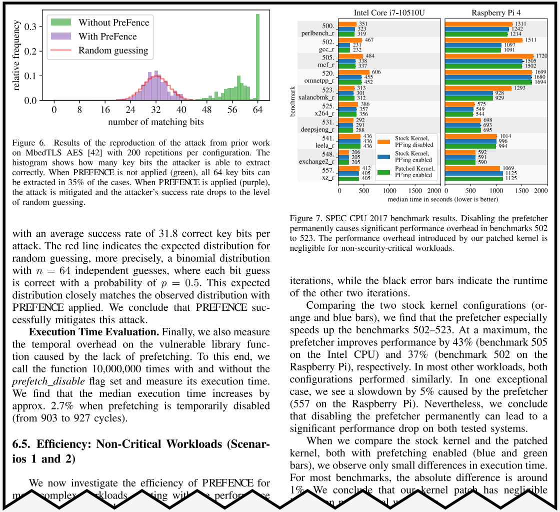


Figure 6. Results of the reproduction of the attack from prior work on MbedTLS AES [42] with 200 repetitions per configuration. The histogram shows how many key bits the attacker is able to extract correctly. When PREFENCE is not applied (green), all 64 key bits can be extracted in 35% of the cases. When PREFENCE is applied (purple), the attack is mitigated and the attacker's success rate drops to the level of random guessing.

with an average success rate of 31.8 correct key bits per attack. The red line indicates the expected distribution for random guessing, more precisely, a binomial distribution with $n = 64$ independent guesses, where each bit guess is correct with a probability of $p = 0.5$. This expected distribution closely matches the observed distribution with PREFENCE applied. We conclude that PREFENCE successfully mitigates this attack.

Execution Time Evaluation. Finally, we also measure the temporal overhead on the vulnerable library function caused by the lack of prefetching. To this end, we call the function 10,000,000 times with and without the *prefetch_disable* flag set and measure its execution time. We find that the median execution time increases by approx. 2.7% when prefetching is temporarily disabled (from 903 to 927 cycles).

6.5. Efficiency: Non-Critical Workloads (Scenarios 1 and 2)

We now investigate the efficiency of PREFENCE for more complex workloads, starting with the performance

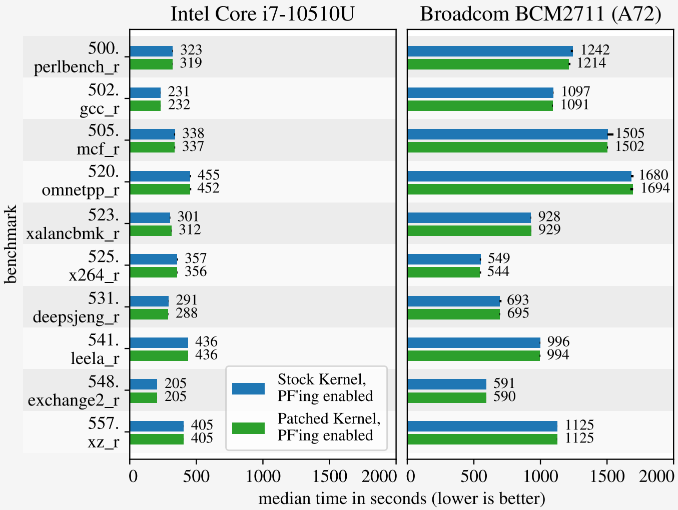
Figure 7. SPEC CPU 2017 benchmark results. Disabling the prefetcher permanently causes significant performance overhead in benchmarks 502 to 523. The performance overhead introduced by our patched kernel is negligible for non-security-critical workloads.

iterations, while the black error bars indicate the runtime of the other two iterations.

Comparing the two stock kernel configurations (orange and blue bars), we find that the prefetcher especially speeds up the benchmarks 502–523. At a maximum, the prefetcher improves performance by 43% (benchmark 505 on the Intel CPU) and 37% (benchmark 502 on the Raspberry Pi), respectively. In most other workloads, both configurations performed similarly. In one exceptional case, we see a slowdown by 5% caused by the prefetcher (557 on the Raspberry Pi). Nevertheless, we conclude that disabling the prefetcher permanently can lead to a significant performance drop on both tested systems.

When we compare the stock kernel and the patched kernel, both with prefetching enabled (blue and green bars), we observe only small differences in execution time. For most benchmarks, the absolute difference is around 1%. We conclude that our kernel patch has negligible

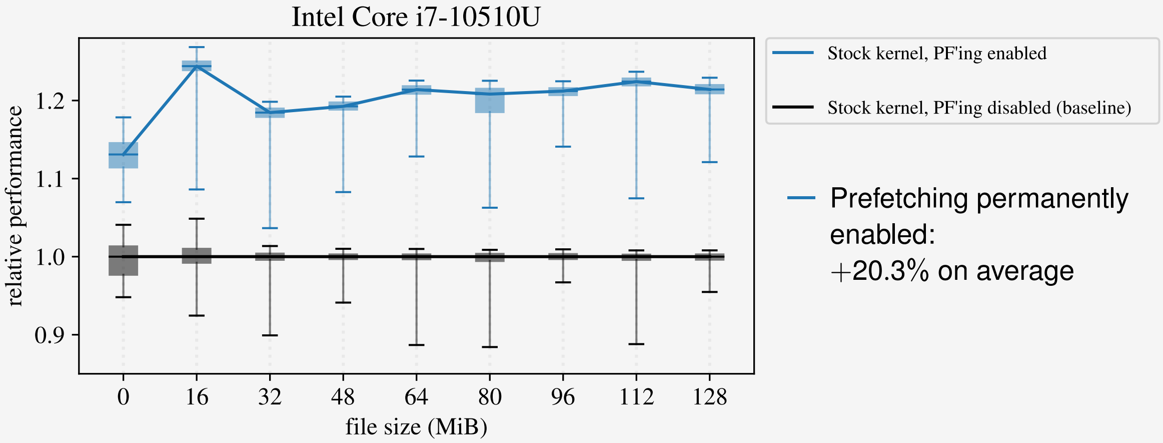
Efficiency: Negligible Overhead on Non-Critical Workloads



SPEC benchmarks perform similarly on stock kernel and patched kernel.

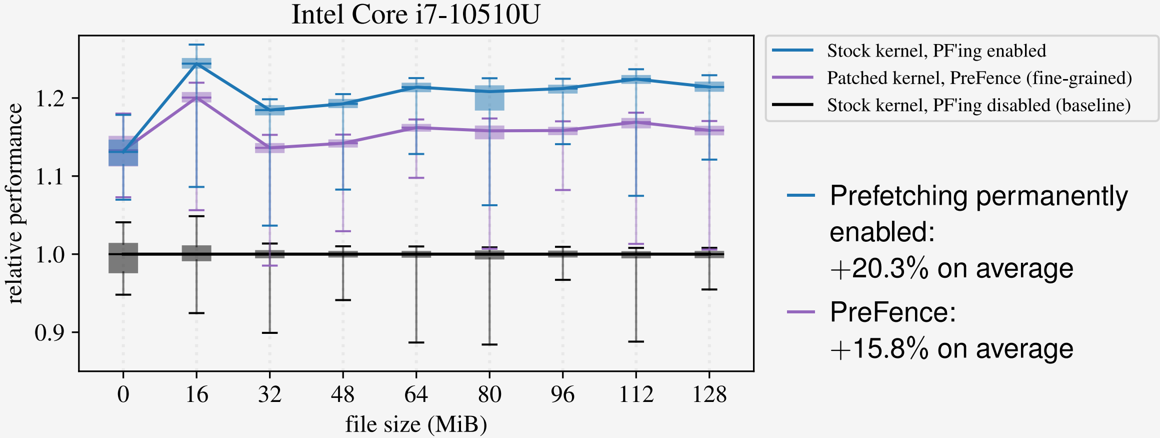
Performance difference around ±1% in most benchmarks.

Efficiency: Bounded Overhead on Critical Workloads



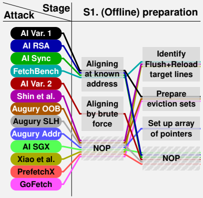
- Prefetching permanently enabled: +20.3% on average

Efficiency: Bounded Overhead on Critical Workloads

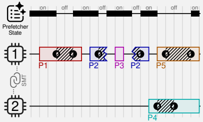


- Prefetching permanently enabled: +20.3% on average
- PreFence: +15.8% on average

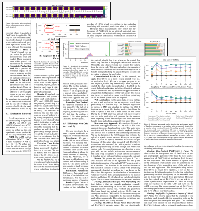
Systematization



PreFence



Evaluation

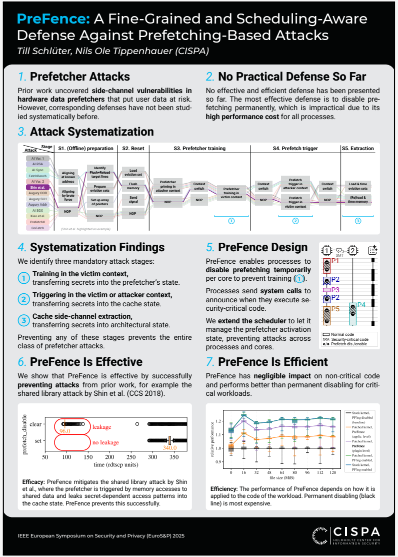


Till Schlüter

✉ till.schlueter@cispa.de

tschlueter.com

github.com/scy-phy/PreFence



PreFence: A Fine-Grained and Scheduling-Aware Defense Against Prefetching-Based Attacks

Till Schlüter, Nils Ole Tippenhauer (CISPA)

1. Prefetcher Attacks

Prior work uncovered **side-channel vulnerabilities in hardware data prefetchers** that put user data at risk. However, corresponding defenses have not been studied systematically before.

2. No Practical Defense So Far

No effective and efficient defense has been presented so far. The most effective defense is to disable prefetching permanently, which is impractical due to its **high performance cost** for all processes.

3. Attack Systematization

4. Systematization Findings

We identify three mandatory attack stages:

1. **Training in the victim context,** transferring secrets into the prefetcher's state.
2. **Triggering in the victim or attacker context,** transferring secrets into the cache state.
3. **Cache side-channel extraction,** transferring secrets into architectural state.

Preventing any of these stages prevents the entire class of prefetcher attacks.

5. PreFence Design

PreFence enables processes to **disable prefetching temporarily** per core to prevent training (1).

Processes send **system calls** to announce when they execute security-critical code.

We **extend the scheduler** to let it manage the prefetcher activation state, preventing attacks across processes and cores.

6. PreFence Is Effective

We show that PreFence is effective by successfully **preventing attacks** from prior work, for example the shared library attack by Shin et al. (CCS 2018).

Efficacy: PreFence mitigates the shared library attack by Shin et al., where the prefetcher is triggered by memory accesses to shared data and leaks secret-dependent access patterns into the cache state. PreFence prevents this successfully.

7. PreFence Is Efficient

PreFence has **negligible impact** on non-critical code and performs better than permanent disabling for critical workloads.

relative performance
1.2
1.1
1.0
0.9
0
16
32
48
64
80
96
112
128
file size (MiB)
Stock kernel, PFing disabled (baseline)
Patched kernel, PreFence (applic. level)
Patched kernel, PreFence (plugin level)
Patched kernel, PFing enabled
Stock kernel, PFing enabled

Efficiency: The performance of PreFence depends on how it is applied to the code of the workload. Permanent disabling (black line) is most expensive.

IEEE European Symposium on Security and Privacy (EuroS&P) 2025

CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

References I

[1] Boru Chen et al. "GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers". In: USENIX Security. 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/chen-boru.

[2] Yun Chen, Lingfeng Pei, and Trevor E. Carlson. "AfterImage: Leaking Control Flow Data and Tracking Load Operations via the Hardware Prefetcher". In: ASPLOS. 2023. DOI: 10.1145/3575693.3575719.

[3] Yun Chen et al. "PREFETCHX: Cross-Core Cache-Agnostic Prefetcher-Based Side-Channel Attacks". In: HPCA. 2024. DOI: 10.1109/HPCA57654.2024.00037.

[4] Jose Rodrigo Sanchez Vicarte et al. "Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest". In: S&P. 2022. DOI: 10.1109/SP46214.2022.9833570.

[5] Till Schlüter et al. "FetchBench: Systematic Identification and Characterization of Proprietary Prefetchers". In: CCS. 2023. DOI: 10.1145/3576915.3623124.

References II

[6] Youngjoo Shin et al. “Unveiling Hardware-Based Data Prefetcher, a Hidden Source of Information Leakage”. In: CCS. 2018. DOI: 10.1145/3243734.3243736.

[7] Chong Xiao, Ming Tang, and Sylvain Guilley. “Exploiting the Microarchitectural Leakage of Prefetching Activities for Side-Channel Attacks”. In: Journal of Systems Architecture 139 (June 2023). DOI: 10.1016/j.sysarc.2023.102877.

This presentation contains icons from (or derived from) the Fluent UI system icons collection, © Microsoft Corporation, MIT License.