ShowTime: Amplifying Arbitrary CPU Timing Side Channels

Antoon Purnal
imec-COSIC, KU Leuven

Marton Bognar
imec-DistriNet, KU Leuven

Frank Piessens
imec-DistriNet, KU Leuven

Ingrid Verbauwhede
imec-COSIC, KU Leuven

ABSTRACT

Microarchitectural attacks typically rely on precise timing sources to uncover short-lived secret-dependent activity in the processor. In response, many browsers and even CPU vendors restrict access to fine-grained timers. While some attacks are still possible, several state-of-the-art microarchitectural attack vectors are actively hindered or even eliminated by these restrictions.

This paper proposes ShowTime, a general framework to expose arbitrary microarchitectural timing channels to coarse-grained timers. ShowTime consists of Convert routines, transforming microarchitectural leakage from one type to another, and Amplify routines, inflating the timing difference of a *single* microarchitectural event to make it distinguishable with crude sources of time.

We contribute several Convert and Amplify routines and show how to combine them into powerful attack primitives. We demonstrate how a single cache event can be amplified so that even the human eye can classify it with 98% accuracy and how stateless time differences as minuscule as 20 ns can be captured, converted, and amplified in a single observation. Additionally, we generate cache eviction sets, both in real-world restricted browser environments and natively using timers with precisions ranging from microseconds to *seconds*. Our findings imply that timer restrictions alone, even when ruthlessly implemented beyond practical limits, provide insufficient protection against CPU timing attacks.

CCS CONCEPTS

• **Security and privacy → Software and application security**; **Systems security**; **Browser security**.

KEYWORDS

CPU side channel; microarchitecture; restricted timers; JavaScript

1 INTRODUCTION

In modern computing systems, programs may affect the execution of other programs through incidental interference in shared hardware components (e.g., caches or computational units). Such interference, predictably, affects the performance of software on multi-tenant systems. However, sharing the processor hardware (i.e., the processor *microarchitecture*) also carries security implications. Indeed, by measuring how long it takes to execute specific actions (i.e., a *timing side channel*), malicious programs can determine the usage patterns of specific microarchitectural components. Therefore, any program with secret-dependent resource utilization unintentionally encodes its secrets in the microarchitecture, exposing it to co-located adversaries. Several attacks manage to exploit this behavior, revealing cryptographic keys [6, 26, 46, 77], operating system secrets [19, 21, 25, 29], or user input [23, 45, 53].

Microarchitectural leakage is often categorized into *stateful* channels, whose effect on the microarchitecture endures for some time after the secret-dependent execution, and *stateless* channels, for which the influence on the microarchitecture disappears as the secret-dependent instructions finish executing. Initially, stateful attacks (e.g., [18, 28, 40, 46, 77]) attracted more attention as they allow the side-channel measurement to happen sometime after the secret-dependent activity. Recently, however, stateless side-channel attacks were also proven to be powerful [4, 7, 12, 47, 69, 78, 79].

Microarchitectural leakage by itself, whether it be stateful or stateless, produces only minuscule timing differences (e.g., 10-100 ns). Therefore, the lion's share of timing side-channel attacks rely on high-precision sources of time, either by consulting existing timer interfaces (e.g., [8, 15, 18, 40, 45, 77]) or by producing fine-grained and monotonically increasing values that correlate with time (e.g., [19, 57, 58]). In response, some platforms disable unprivileged access to high-precision timers [36]. Even stronger, there have been academic proposals [35, 41, 64] to orchestrate computing environments that eliminate all high-precision timing sources. Similar measures are currently deployed in modern browsers [16, 17, 44, 68, 72].

Without fine-grained timers, one option is to repeatedly trigger the leak (*multi-shot amplification*) [42, 54, 59, 73]. However, this requires a deterministic repetition of the leak, which is not generally possible. Moreover, timing differences typically accumulate slowly. In contrast, the website `leaky.page` [56] performs a sequence of memory accesses that, conditioned on the presence of a single target memory access (*single-shot amplification*), accrue measurable timing differences (e.g., 100 µs). While powerful, this technique can only expose the presence of same-process memory accesses.

At this time, it is unclear whether several state-of-the-art microarchitectural attacks, e.g., those that obtain privilege escalation [13, 22, 33] or extract secrets across processor cores [40, 45, 77], still pose a threat without high-precision timers. A key unsolved problem is finding *eviction sets*, i.e., sets of memory addresses contending for capacity in the last-level cache (LLC) [40, 50, 51, 67, 76]. Moreover, stateless channels appear challenging to amplify, as they are inherently short-lived. Therefore, we ask:

Can cross-core side-channel attacks be mounted with low-precision timers? Can stateless side channels be amplified at all? What are the limits to single-shot microarchitectural amplification?

In this paper, we show that microarchitectural attacks are not foundationally thwarted by restricted timing sources. We present ShowTime, a generic framework to produce considerable timing differences from fine-grained leaks, regardless of their source.

AsiaCCS '23, July 10–14, 2023, Melbourne, Australia

ShowTime composes two phases of independent interest. First, the CONVERT phase transforms an initial microarchitectural leak to make it amenable to enter the second phase, AMPLIFY, which produces a large timing difference depending on its input state.

For the AMPLIFY phase of ShowTime, we develop novel instruction sequences that exhibit considerable differences in runtime, depending on a single microarchitectural state difference. First, we generalize the `leaky.page` amplifier [56] for use in ShowTime, permitting the single-shot amplification of additional events, such as differences in load order or cache line invalidations. We show how to increase its *amplification ratio*, i.e., the ratio between slow and fast executions of the amplifier, from 1.3 to 2.3. Additionally, we develop robustness measures to increase the maximal time difference it can reliably produce, from 500 µs to 5 ms. Second, we discover a powerful single-shot amplifier for adversaries with native code execution. Its amplification ratio exceeds 10, meaning that it takes, e.g., less than 1.1 ms to produce a difference of 1 ms. Due to its unprecedented robustness, it can produce timing differences far beyond any timer coarseness imposable in practice.

For the CONVERT phase of ShowTime, we contribute several conversion techniques, relying on well-known CPU behavior like out-of-order execution, cache line back-invalidation, and thread-level parallelism. They make it possible to convert (single-shot, potentially cross-core, potentially stateless) microarchitectural leaks to make them attacker-visible, attacker-amplifiable, and both.

To show how ShowTime fares with stateless leaks, we develop a proof-of-concept attack to expose port contention on *another processor core* through its delaying effect on following instructions. Though this secret-dependent delay does not exceed 20 ns, ShowTime can capture, convert and amplify its presence or absence. We also demonstrate that ShowTime can be used to reveal information on the CPU frequency at a given point in time, which is an inherently stateless microarchitectural context that has recently been shown to produce severe leakage [39, 70].

Exploring the limits of microarchitectural amplification, we find that our strongest amplifier can generate time differences so enormous that the human eye can classify a single initial cache hit or miss with more than 98% accuracy. In addition, we find eviction sets for the LLC using the Unix Epoch, an excessively crude "timer" reflecting the number of seconds elapsed since January 1, 1970.

We also construct LLC eviction sets in JavaScript with a 100 µs timer, which is the most restricted scenario in the latest Chrome release. The median execution time is 25 s, for an accuracy of 70%. These results show that the ShowTime convert-and-amplify strategy is also successful from the browser, which is a restricted execution context where timers are already limited in practice.

Contributions. Our main contributions are the following:

- We provide a framework to expose fine-grained timing leaks of arbitrary type to coarse-grained timing sources.
- We develop robust amplifiers capable of producing large time differences from unique microarchitectural events.
- We show how to reliably convert activity in one microarchitectural component into controlled activity in another.
- We evaluate ShowTime for cross-core stateless attacks, frequency measurements, and eviction set construction.

We disclosed our findings to Intel and Google.

Availability. To facilitate the reproduction of our research, artifacts are available at

https://github.com/KULeuven-COSIC/ShowTime

2 BACKGROUND

Cache Hierarchy. Modern processors consume and produce data faster than main memory technology can provide and accept it. To overcome this issue, processors feature a *cache hierarchy*; a series of successively smaller and faster pieces of on-chip memory. Typically, caches are implemented as a two-dimensional array of *cache lines*. This array is indexed into *sets*, to which cache lines are mapped based on their memory address. Lines mapped to the same set are called *congruent*, and the number of congruent lines mapped to the same set is the cache's associativity (or its number of *ways*).

The cache hierarchy on Intel processors comprises three levels. Each core has its own L1 and L2 cache, the two fastest and smallest levels. The L3 or *last-level cache* (LLC) is shared between all CPU cores. Most Intel LLCs abide by an *inclusive* policy, stating that all cache lines in L1/L2 necessarily have a copy in the LLC.

In the event of a *cache miss*, i.e., the cache does not contain the requested memory address, the next level cache is consulted, cascading all the way to main memory in case the request triggers a cache miss in all levels. To install the new line in the cache set, one of its existing entries is selected to be *replaced* (or *evicted*), and the state machine that governs this selection is the *replacement policy*.

Sometimes, the programmer or compiler may want to instruct the processor to fetch specific data before it is used. On Intel processors, several so-called *prefetch* instructions exist for this purpose.

Cache Attacks. The presence of a shared cache hierarchy implies that processes affect each other's runtime through competitive use of the cache space. This introduces a *timing side channel*. For instance, a malicious process occupying an entire cache set can determine, by measuring the access latency of its own cache lines, whether one of them was evicted by the activity of another process. Such an attack is known as Prime+Probe [40, 46]. Recently, a more precise version, Prime+Scope [51], was proposed, which concentrates the contention with the victim into a single cache line. A key prerequisite for Prime+Probe-style techniques is to find *eviction sets*, i.e., addresses that map to the same set in the target cache.

Other Microarchitectural Leakage. While the cache hierarchy has been the most prominent target for timing attacks, CPU microarchitectures feature several other components that expose processes to the metadata leakage of other processes. Any component competitively shared between potential attacker and victim processes can be the target of a timing attack. Some of these components are core-private, e.g., L1 caches [46], execution ports [4, 7], TLBs [18], and fetch/decode units [62], implying that an attacker needs to obtain core-level co-location with their victim to mount an attack. Other components, e.g., DRAM row buffers [49], and on-chip interconnects [12, 47, 69], are competitively shared across cores, relaxing the co-location requirements for the attacker.

Out-of-Order Execution. To maximally make use of available hardware resources, modern processors implement *out-of-order execution*. This feature exploits instruction-level parallelism, allowing independent instructions to execute as soon as their operands are available, instead of strictly adhering to the order specified by the

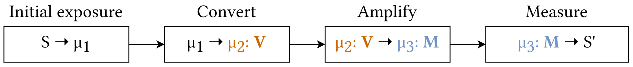


Figure 1: ShowTime framework.

program's (inherently serial) software description. Out-of-order execution itself may be a source of hardware vulnerabilities [38].

3 SHOWTIME

3.1 Threat Model

We consider an attacker with unprivileged code execution. The only timing sources available to the attacker are *coarse-grained timers*, i.e., their granularity (e.g., 5 µs, 100 µs, or 100 ms) is several orders of magnitude larger than the timing variations the attacker intends to measure (e.g., 10 ns). We explicitly do *not* assume that the attacker runs on the same CPU core as the victim, that huge memory pages are available, or that the CPU frequency is fixed.

3.2 General Framework

Figure 1 shows the ShowTime cascade for a general microarchitectural side-channel attack. The data flow starts from the initial exposure of secret architectural data (S) to the microarchitectural context (μ_1). At the end, the attacker reconstructs the secret data by measuring a transformed context. For this reconstruction to be possible, the final context needs to have two properties:

- Visible (**V**): the leakage is present in a component shared with the adversary or observable through an explicit interface [14].
- Measurable (**M**): the leakage is strong enough to be picked up by the measurement source. Being measurable implies being visible.

ShowTime aims to achieve measurability for arbitrary initial exposure types with Convert and Amplify phases, which we now describe briefly. In practice, some steps may be repeated, skipped, or reordered.

3.2.1 *Initial Exposure.*

The initial encoding of secret data into the microarchitecture can be categorized according to different criteria:

- Visible (**V**) or invisible: the latter may be the case for leaks in core-private resources such as execution ports [4, 7].
- Unintentional (in a side-channel attack), or intentional (in a covert channel or a transient execution attack [31, 38]).
- Stateful or stateless, i.e., with persisting (stateful) or ephemeral (stateless) interference in the microarchitectural context.

3.2.2 *Convert.*

The Convert step translates exposure in one microarchitectural component into exposure in another. This can be required for multiple reasons. If the initial exposure is not visible (**V**), it can be transformed to a visible encoding in the microarchitectural context. If the initial leakage is not measurable (**M**) and cannot be directly amplified (e.g., it is stateless), it can first be transformed into an amplifiable (e.g., more persistent [3, 5]) state.

Similar to the initial exposure, conversions can be unintentional or intentional. Unintentional conversions can occur through gadgets in the victim code or implicitly through processor hardware features (e.g., dynamic voltage and frequency scaling (DVFS) converts power differences into frequency differences [70]).

3.2.3 *Amplify.*

Leakage that is visible (**V**) but not measurable (**M**) requires amplification before it can be decoded. An amplifier is a piece of code whose execution time is deliberately made sensitive to a specific difference in the microarchitectural context.

Prior work mostly focuses on constructing *multi-shot* amplifiers, which repeatedly trigger an identical initial exposure [42, 54, 59, 73]. However, attackers cannot always force the victim to repeatedly execute with the same inputs. Moreover, while existing amplification techniques are theoretically capable of achieving arbitrary time differences, it is unclear whether they remain applicable in practice as timing sources get restricted even further, e.g., to 100 ms [63].

In this work, we focus on single-shot amplification. From here on, *amplifier* and *amplification* refer to single-shot methods.

3.2.4 *Measure.*

The final step in ShowTime is to read out the side-channel information in the architectural domain to reconstruct (S') the secret (S). For timing side channels, the architectural value is typically obtained by reading a monotonically increasing value before and after executing the target code. This value can either be readily accessible (e.g., `rdtsc` in x86 or `performance.now()` in JavaScript), or implemented by the attacker [34, 55, 57, 58]. The difference can then be thresholded to recover the initial secret.

Alternatively, the thresholding can also be a part of the measurement, e.g., by testing whether the target executes slower or faster compared to an action with a known execution time [60]. Other software-accessible measurement interfaces include hardware transactional memory [14, 20], on-chip power consumption monitors [37], and the CPU frequency [70] manager. However, such direct interfaces can be or have been disabled [27] or weakened [37]. In this work, we focus on timing side channels.

Restrictions in Browsers. In response to Spectre [31], browsers limit JavaScript features that can be correlated to the precise passage of time, especially in combination with interacting with other websites [16]. Websites can opt into these features by explicitly setting two HTTP response headers, enabling cross-origin isolation.

In current versions of Chrome (≥ 92) [16] and Firefox (≥ 79) [43], `SharedArrayBuffer` is one of these restricted features, as it can be used for constructing a precise timer [57]. In Chrome, the granularity of `performance.now()` is limited to 5 µs on isolated and 100 µs on non-isolated sites [16, 17, 68]. In Firefox [44] and Safari (WebKit) [72] `performance.now()` is further degraded to a precision of 1 ms. Even before Spectre, the Tor Browser limited its precision to 100 ms [63].

4 SINGLE-SHOT AMPLIFICATION

This paper studies *single-shot amplifiers*, i.e., unprivileged programs whose execution time depends on a single difference in a microarchitectural context. Not relying on multiple victim code invocations makes these amplifiers applicable in more attack scenarios.

Amplifier Quality. The capabilities of a single-shot amplifier can be quantified by different metrics. Its *amplification ratio* ($A \geq 1$) defines the ratio between the amplifier's slow and fast execution times. The *maximal output timing difference* Δ is the absolute difference between the slow and fast times the amplifier can reliably produce. As we will see, although some amplifiers are theoretically capable of producing arbitrary timing differences, in practice they seem to degrade when a specific timing difference is reached. Finally,

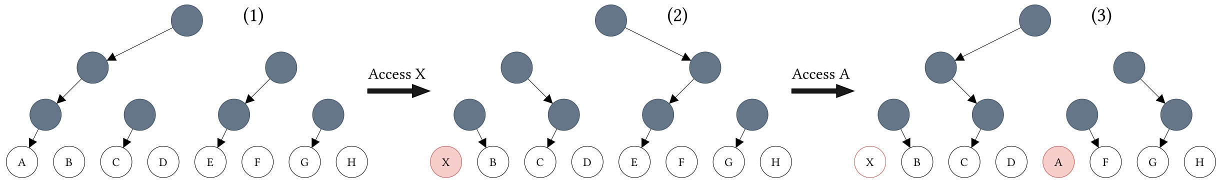


Figure 2: Amplifier based on the L1 replacement policy state.

the initial microarchitectural contexts the amplifier can capture determines how widely *applicable* the amplifier is.

4.1 Amplification Using the L1 PLRU

PLRU Replacement Policy. Modern Intel processors have an 8-way set-associative L1 data cache, which implements an approximation of the least-recently-used (LRU) replacement policy, dubbed PLRU (for pseudo-LRU). Conceptually, cache lines are organized as the leaves of a balanced binary tree structure (cf. Figure 2). The state of each node of the tree is carried by one state bit, which can be thought of as an arrow, pointing to one of its two children.

On a cache *hit*, i.e., when the requested line can be served directly from the L1d cache, the arrows at each node along the path from the root to the cache line are set to point away from this line. On a cache *miss*, the requested line is loaded into the cache. The line to be replaced (or *evicted*) is selected by following the direction of the arrows from the root of the tree to one of the leaves. Then, similarly to a cache hit, the direction of all traversed arrows is set to point away from the newly inserted line. The line that is currently cached, but is next to be evicted, is said to be the *eviction candidate*.

Basic PLRU Amplification. `leaky.page` [56] proposes a single-shot amplification technique that captures an *L1 Eviction* event, e.g., the secret-dependent eviction of an attacker line A by a line X that maps to the same L1 set. In particular, their PLRU amplifier repeatedly traverses a sequence of memory loads mapping to specific cache lines (which, to prevent the secret-dependent state from being destroyed, does not include the line X itself). This traversal exhibits a different ratio of L1 hits and misses conditioned on the secret-dependent eviction of A (the *L1 Eviction* event). Given that L1 cache misses take longer to resolve than L1 cache hits, the different hit/miss pattern gives rise to *fast* and *slow* instances. By repeating the traversal until the time difference between the fast and the slow pattern is larger than the timer granularity Δ, the occurrence of the *L1 Eviction* event can be revealed with a low-resolution timer.

Concretely, consider cache lines `A`–`H`, which all map to the same L1d set. Accessing the preparation pattern `load(AECGBFDH)` (i.e., a load to A, then to E, etc.) produces the initial state of the PLRU tree as in Figure 2, or one that is equivalent to it, up to permuting the two children of each node. Note that this is only guaranteed as long as none of the lines `A`–`H` are cached prior to the pattern, which is a prerequisite that can be fulfilled by evicting the relevant L1 set. To ensure that the processor does not reorder the loads of line `A`–`H`, they are serialized through a data dependency [56].

To describe the traversal pattern, we adopt a compact notation. The format is `traverse(*)`$_{B^n}$, where `*` is one iteration of the base pattern, and B^n implies line B is accessed once every n accesses of the base pattern. Therefore, `traverse(AECGFDH)`$_{B^4}$ is short for the repeated traversal of $\texttt{load(AEC\underline{B}GFD\underline{B}HAE\underline{B}CGF\underline{B}DHA\underline{B}ECG\underline{B}FDH\underline{B}..)}$.

Table 1: Traversal and refresh patterns (novel in bold), along with the sequence of hits (H) and misses (M) they generate. Accesses corresponding to line B are underlined.

Traversal	Hit/Miss (1)	Hit/Miss (2)	Type
`traverse(AECGFDH)`$_{B^4}$	$\texttt{HHH\underline{H}HHH\underline{H}..}$	$\texttt{MMM\underline{H}MMM\underline{H}..}$	Distance 1
`traverse(AECGFDH)`$_{B^3}$	$\texttt{HH\underline{H}HH\underline{H}..}$	$\texttt{MM\underline{H}MM\underline{H}..}$	Distance 2
`traverse(AECGFDH)`$_{B^2}$	$\texttt{H\underline{H}H\underline{H}..}$	$\texttt{M\underline{H}M\underline{H}..}$	Distance 3

Refresh	Type
$\mathbf{\texttt{load(012\underline{B}345\underline{B}ECG\underline{B}FDH\underline{B})}}$	Distance 1
$\mathbf{\texttt{load(01\underline{B}23\underline{B}4\underline{B}5\underline{B}EC\underline{B}GD\underline{B}F\underline{B}H\underline{B})}}$	Distance 2
$\mathbf{\texttt{load(0\underline{B}1\underline{B}2\underline{B}3\underline{B}E\underline{B}G\underline{B}F\underline{B}H\underline{B})}}$	Distance 3

Due to the PLRU replacement policy in the L1d cache, all accesses are L1 hits if the *L1 Eviction* event did not occur, and only 25% of them are hits if it did occur. Figure 2 explains why. In case the *L1 Eviction* did not occur, the cache remains in state (1). Naturally, as every element of the traversal pattern is still in L1, all accesses will be hits. If the event did occur, A was evicted from the cache and replaced by X (2). At the same time, E became the next eviction candidate. In the first step of the traversal, we access A, which, since it was evicted, results in a cache miss. Since E is the new eviction candidate, E will be replaced by A, and C becomes the eviction candidate (3). In the next step, we access E, but since it was just evicted, it will result in another miss, etc. The repeated access to B serves to prevent X from being evicted, without accessing X itself.

Improving the Amplification Ratio. To enhance the power of the PLRU amplifier, we propose to perform the traversal with addresses that are congruent in L2. This automatically implies congruence in L1 as well. The traversal patterns remain the same. However, the penalty for the slow pattern becomes larger, as some of the cache misses need to be served from the LLC instead of L2. We also considered traversing LLC-congruent lines but did not observe an additional penalty compared to L2-congruent lines.

Increasing Robustness. Consider when a competing L1d access to the same set occurs, e.g., by another process running on the same physical core. If, at any point, line X or B is evicted, the amplifier no longer works. The original `traverse(AECGFDH)`$_{B^4}$ sequence is not very robust against this; if another access to the L1 set occurs before any of the accesses to B, X is evicted (i.e., once every four accesses, X is the *first* in line to be evicted in case of a cache miss). Therefore, Table 1 contains more robust sequences where X is at worst *two* (distance-2) or *three* (distance-3) cache misses away from being evicted. This is obtained by accessing B more frequently and comes at the cost of (slightly) decreasing the amplification ratio.

Table 2: Amplifying other events in the L1d cache. The adaptor modifies the state difference to match the one in Figure 2, such that identical traversal patterns can be used.

Amplify	Initialize	Event (option 1/2)	Adaptor
L1 Eviction	`load(AECGBFDH)`	`load(X)` / ⊥	⊥
L1 Reordering	`load(AECGBFDH)`	`load(DH)` / `load(HD)`	`load(XFHB)`
L1 Back-Invalidation	`load(AECGBFDH)`	`invalidate(E)` / ⊥	`load(XFHB)`

As an optional robustness measure against degradation of the L1 state due to noise, we also propose to *refresh* it periodically. That is, we periodically evict the L1 set with additional lines 0-5 that map to the same set, without affecting the presence or absence of lines X and B. This can occur with the refresh patterns in Table 1. Note that refreshes should only be repeated once every so many traversals, e.g., 128, and hence are negligible for the execution time.

Expanding Measurable Events. We now discuss how the PLRU single-shot amplifier can be generalized to be conditioned on other initial microarchitectural contexts relating to the L1 data cache, i.e., *L1 Reordering* and *L1 Back-Invalidation* (cf. Table 2).

L1 Reordering is captured in the following manner. The L1 PLRU state is prepared as before (i.e., `load(AECGBFDH)`). Recall that line A is the eviction candidate after the preparation. We aim to capture the load order of lines D and H. If D is accessed before H, A remains the eviction candidate. If, instead, H is accessed before D, E becomes the eviction candidate. Now consider another access to line X, serialized to happen after both loads. It evicts either line A or E, depending on the load order of D and H. Then, after accessing a short adaptor sequence (Table 2), traversing the original *L1 Eviction* pattern exhibits the same hit/miss pattern as the *L1 Eviction* event.

L1 Back-Invalidation is captured as follows. The state is prepared as before (i.e., `load(AECGBFDH)`). Line A is the eviction candidate. Consider the event where line E is evicted from the LLC. To satisfy the LLC inclusion property, this triggers a back-invalidation of line E in L1. Now consider another access, to line X, happening after this potential back-invalidation. If the invalidation has occurred, X takes the place of E, since the L1 replacement policy favors filling empty ways. If it did not occur (⊥), X evicts A. Again, a short adaptor sequence makes it behave like the original *L1 Eviction* pattern.

Amplify: L1 PLRU.
PLRU can capture reorderings and invalidations. L2-congruent lines increase the amplification ratio, and distance-2/3 sequences boost robustness.

4.2 Amplification Using prefetchNTA

Non-Temporal Prefetch on Intel x86. `prefetchNTA` is a software prefetch instruction with a *non-temporal hint*, communicating to the processor that this data will not be used multiple times. Its microarchitectural behavior on Intel CPUs was previously studied by Guo et al. [24]. Importantly, lines cached using this instruction are treated differently by the LLC replacement policy. For details on this replacement policy, we refer the reader to prior work [1, 10, 24, 66].

For our purposes, three generic properties are relevant. First, for lines that are not cached, `prefetchNTA` performs a cache line fill in the LLC, but with the highest age, making it very likely to become the eviction candidate. Second, prefetching lines that are already cached in the LLC does *not* affect their LLC replacement policy state. Third, `prefetchNTA` takes a (much) longer time to execute for lines in memory than for those in the cache.

```
.rept 1000  ; repeat at will
    mfence
    prefetchnta(A)
    mfence
    prefetchnta(B)
.endr
```

Listing 1: Prefetch-based amplifier.

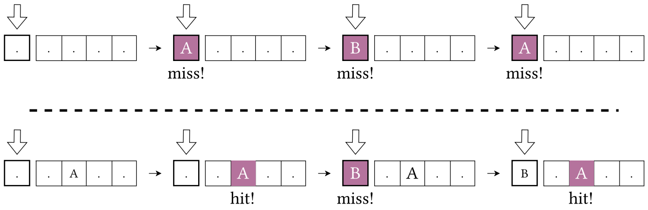


Figure 3: Working principle of the prefetch amplifier (LLC).

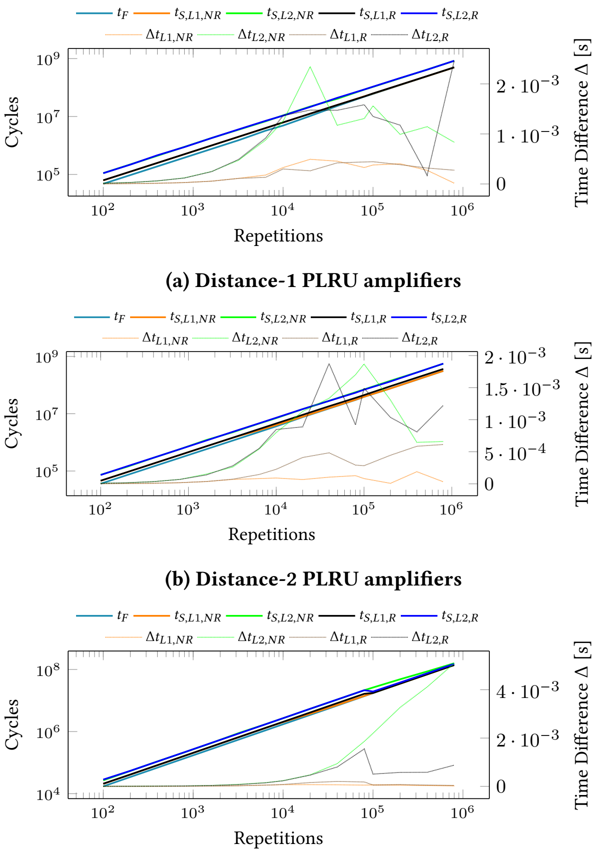
Technique. Figure 3 shows the working principle of the prefetch-based amplifier. The initial microarchitectural context that conditions the amplifier is the LLC caching state of an attacker line A. Assume that the attacker also has access to a line B, which is not cached but maps to the same LLC set as A. The amplifier is a repeated alternating `prefetchNTA` of lines A and B, serialized with `mfence` instructions to maintain their execution order (Listing 1).

Consider the case where line A is not cached, shown on the top half of Figure 3. The `prefetchNTA` of line A caches it in the LLC as the eviction candidate (indicated by the empty arrow). The `prefetchNTA` of line B evicts A, and installs B as the new eviction candidate. As the pattern is repeated, every prefetch is served from memory, slowing down the execution.

Now, consider the case where line A is cached (but is not the eviction candidate). The first `prefetchNTA` of A is fast and does not affect its replacement state. Although the first `prefetchNTA` of B is slow, it caches B in the LLC as the eviction candidate and, importantly, does not evict A. All future `prefetchNTA`s of A and B are fast, as both lines remain cached without evicting each other.

Robustness. The fast and slow instances of the prefetch amplifier share the invariant that there is always a prefetched attacker-chosen line in the cache. Therefore, if there are spurious cache line fills (i.e., *noise*) in the LLC set, it is likely that a prefetched line is evicted. In neither of the fast or slow instances does this destroy the state difference needed to keep the amplifier functional. In the fast case, the spurious access will evict B from the LLC, which will make it be loaded from memory *once*, after which the pattern can continue, as A is still cached normally. In the slow case, the spurious access will evict either A or B from the LLC but, regardless of which one, the next prefetch would have been slow anyway.

Amplify: Non-Temporal Prefetch.
Quick LLC eviction enables robust single-shot amplification.



(a) Distance-1 PLRU amplifiers

(b) Distance-2 PLRU amplifiers

(c) Distance-3 PLRU amplifiers

Figure 4: Performance of L1 PLRU amplifiers. The subscripts in the legend denote whether L1- or L2-congruent addresses are used and whether there is a refresh (R) or not (NR). On some subfigures, $t_{s,L*,NR}$ and $t_{s,L*,R}$ may overlap.

4.3 Evaluation

PLRU Amplifiers. Figure 4 depicts the fast (t_F) and slow (t_S) traversal times of the L1 amplification patterns as a function of the number of repetitions, along with the time difference they produce. We do not evaluate the *L1 Eviction*, *L1 Reordering*, and *L1 Back-Invalidation* amplifiers separately, since they have identical performance. For each data point, we consider 100 runs of 100 iterations and take the median over all runs. When refresh patterns are enabled, they are accessed once every 1024 traversals (with distance 2). The amplification ratio is constant for small Δ, but as Δ increases, noise accumulates and the amplification ratio degrades until supposedly fast and slow traversals are no longer distinguishable. However, amplifiers vary in their resilience to degradation.

Prefetch Amplifier. Figure 5 shows the median traversal times using the prefetch amplification method on the Intel Core i7-7700K. The initial amplification ratio exceeds one order of magnitude, which it maintains until roughly 1 billion cycles, after which it declines. Due to its robustness, the amplifier is able to produce time differences of several hundreds of ms from a single initial difference.

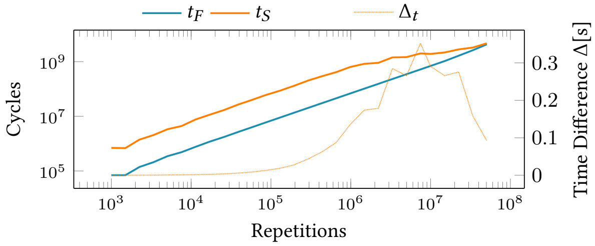


Figure 5: Performance of the prefetch-based amplifier (median of 10 batches of 100 runs per data point).

Table 3: Comparison of single-shot amplifiers.

Source	Amplifier	Single-Shot	A	Max. Δ
`leaky.page` [56]	L1 PLRU (L1-congr., dist-1)	✓	1.3	≈ 500 µs
This Work	L1 PLRU (L2-congr., dist-1)	✓	2.3	2.4 ms
This Work	L1 PLRU (L2-congr., dist-2)	✓	2.0	1.8 ms
This Work	L1 PLRU (L2-congr., dist-3)	✓	1.6	5.1 ms
This Work	prefetchNTA	✓	10.1	350 ms

Comparison. Table 3 collects the best amplifier instances of each type, along with their amplification ratio A and maximal output difference Δ. It confirms that traversing L2-congruent lines, instead of L1-congruent lines, produces a larger time difference. For the L1-congruent distance-1 amplifier [56], our best implementation achieves a maximal output difference of 500 µs. For the distance-3 sequences, we observe output differences up to 1.5 ms for L1-congruent addresses, and 5 ms for L2-congruent addresses.

In Figure 4, periodic refreshes appear to increase the robustness of the L1 sequences, but no such effect is visible for the L2 sequences.

Measurement Rate. The rate at which timing measurements can be performed for a timing source of granularity Δ is determined by the amplification ratio A of the amplifier. With A defined as the ratio $\frac{t_S}{t_F}$, and $t_F - t_S = \Delta$, this implies that $t_F = \frac{\Delta}{A-1}$ and $t_S = \frac{A \cdot \Delta}{A-1}$. As an example, to produce a timing difference of $\Delta = 100\,\mu s$, a slow measurement for the `leaky.page` PLRU amplifier ($A = 1.3$) takes $\approx 4.3\Delta = 430\,\mu s$. It takes $\approx 1.8\Delta = 180\,\mu s$ for our best PLRU amplifier, and $\approx 1.1\Delta = 110\,\mu s$ for our prefetch-based amplifier. Note that these estimates are only valid for the regimes in which A is constant (and hence independent of Δ). If amplifiers are used beyond their robustness capabilities, they may not even produce any meaningful timing difference anymore (cf. Figure 4).

Practical Considerations. The prefetch-based amplifier relies on the x86 `prefetchNTA` instruction and on Intel's implementation choice of marking prefetched lines for quick eviction from the inclusive LLC [24]. A similar amplifier may be devised to exploit the LLC replacement policy (cf. [5, 9, 10]) without a prefetch instruction, at the cost of a lower amplification ratio. The L1-based amplifiers do not require the exposure of specific instructions and can hence be used in restricted environments (cf. [56] and Section 6.3).

The prerequisites for our single-shot amplifiers are met for a wide range of Intel processors [24, 56]. However, other CPU families are not guaranteed to satisfy them. Still, the existence of single-shot amplification demonstrates that innocuous implementation decisions invalidate high-level security properties that are, at the

```
dep = prepare-uarch()

// first leg                    // second leg
dep1 = secret-delay(dep)        dep2 = fixed-delay(dep)
dep1 = instr-1(dep1)            dep2 = instr-2(dep2)

race-end(dep1, dep2)
```

Listing 2: Time to Order conversion.

surface, completely unrelated. We leave an exploration of single-shot amplifiers in other processor families to future work.

5 CONVERTING CPU SIDE CHANNELS

In this section, our objective is to convert side channels of interest to state differences that are amenable to single-shot amplification.

5.1 Back-Invalidation

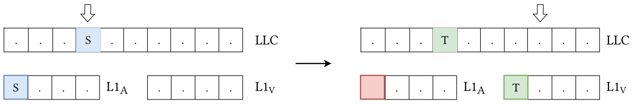


Figure 6: Conversion based on CPU back-invalidation logic. If a line (S) gets evicted from the LLC (by T), all copies of S in the core-private caches get invalidated.

The first technique uses back-invalidation, a deterministic microarchitectural behavior on processors with inclusive LLCs. When a cache line is evicted from the LLC, it is automatically invalidated in the L1 and L2 caches to preserve the inclusiveness invariant. Therefore, the CPU back-invalidation logic produces an implicit conversion from the LLC caching status to an *L1 Back-Invalidation* event, which is amenable to single-shot amplification (cf. Section 4).

With this technique, memory accesses to addresses that map to the monitored LLC set produce the invalidation of a fixed and predictable line in L1. Note that this conversion immediately implies the single-shot amplification of the cross-core Prime+Scope [51] cache attack, which infers LLC activity through the invalidation of a specific line (i.e., *the scope line*) from the L1 cache.

Convert: CPU Back-Invalidation.
LLC evictions automatically produce *L1 Back-Invalidation* events.

5.2 Time to Order

The second conversion technique, Time to Order, exploits the out-of-order execution of instructions on modern processors. Instructions that do not have *data hazards*, i.e., data dependencies on architecturally older instructions, may be executed ahead of these older instructions. Therefore, in an out-of-order processor, the execution order of instructions depends on the time it takes for their dependencies to resolve. As a result, well-designed instruction sequences can encode the latency of specific instruction paths into the execution order of instructions that depend on these paths.

Concretely, as in Listing 2, consider an execution race between two independent legs, which are orchestrated to start at the same time, i.e., through a shared data dependency on another instruction (or the preparation step `prepare-uarch`). One of the legs has a secret-dependent latency, i.e., it contains an operation for which we want to expose the execution time. The other leg has a fixed latency, implemented as an instruction sequence with a fixed execution time (e.g., a sequence of data-dependent multiplications). The length of the fixed-delay sequence is chosen such that the relative execution order of the final instructions of the two legs (resp. `instr-1` and `instr-2`) depends on the latency of the secret-dependent operation. If, in a later stage, the execution order of `instr-1` and `instr-2` can be exposed, it reveals whether the secret-dependent latency is above or below the threshold determined by the fixed-delay leg.

Table 4: Time to Order for our single-shot amplifiers.

Amplifier	prepare-uarch	secret-delay	instr-1	instr-2
L1 Reordering	load(AECGBFDH)	any	load(D)	load(H)
Prefetch	evict(A)	any	load(A)	prefetchNTA(A)

In short, Time to Order turns a timing difference into a difference in the execution order through a microarchitectural race condition. In principle, the timing difference at the *input* (i.e., the event that determines the length of the variable-time leg) can be of arbitrary type. We now show how an instruction ordering at the *output* is amenable to single-shot amplifification. Depending on the preparation and the choice of `instr-1` and `instr-2`, time differences can be cascaded to L1 (cf. Section 4.1) or the LLC (cf. Section 4.2).

5.2.1 *Conversion to L1 Caching Status.*

Table 4 shows how Time to Order can convert a time difference into an *L1 Reordering* event. The microarchitectural state is prepared by filling the PLRU tree as in Section 4.1, and the instructions at the end of each leg are simply the loads as described for the *L1 Reordering* amplifier.

There is no explicit restriction on the type of `secret-delay` that can be converted with Time to Order. Therefore, it is more generally applicable than the back-invalidation conversion (Section 5.1), which has the benefit of being deterministic. A relevant event to capture and convert into the L1 PLRU state is the presence of an LLC hit or miss. Indeed, properly wielding this conversion (see Section 6) reinstates the capability of constructing LLC eviction sets in the browser [45, 67] using the L1 PLRU amplifier, as well as cross-core [40, 45, 51] and cross-process microarchitectural attacks.

Convert: Time to Order (L1).
Encodes a time difference into the L1 PLRU replacement policy.

5.2.2 *Conversion to LLC Caching Status.*

As a conversion to LLC caching status, Table 4 shows a simple Time to Order instance. The instructions at the end of the legs are, respectively, a `prefetchNTA` and a regular load for the same cache line. If the prefetch comes first, the line becomes the eviction candidate. If the load comes first, it (generally) does not. The resulting LLC state difference can directly be amplified using the `prefetchNTA` sequence (cf. Listing 1).

Convert: Time to Order (LLC).
Encodes a time difference into a line's LLC caching status.

The Time to Order primitive is versatile. With some profiling, seemingly unrelated input side channels (stateless or otherwise) can be converted into LLC/L1 state changes, provided that a race can be found for which the outcome reliably depends on the initial leakage type (e.g., cache line status). Other initial leakage types include time-varying instructions, port contention [4, 7, 55], branch predictor state [15], ROB contention [3, 74], DRAM contention [49] and LLC interconnect contention [12, 47, 69]. We cover some of these examples as case studies in Section 6 but leave a full exploration of all amplifiable CPU side channels to future work.

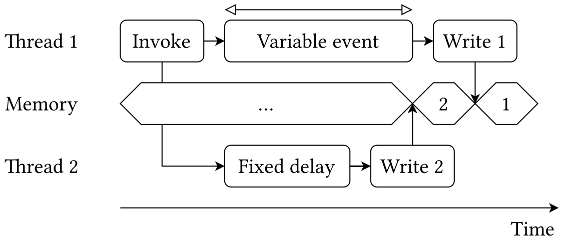


Figure 7: Architectural reordering.

5.3 Architectural Reordering

The final contribution of this section is *architectural reordering*, a novel integrated conversion and measurement routine (cf. Figure 7). To preserve the semantics of a given instruction stream, the processor always executes store operations to the same address in program order. However, no such guarantees exist for stores in *parallel threads*. By conditioning the order of store instructions in different threads on a timing difference, an attacker can directly produce an architectural state change without any timing source.

Accuracy. We use architectural reordering to infer the cache state of a line (causing a hit or a miss in L1). We perform 100 runs of 10,000 iterations for a random initial state. The median accuracy is 100% for hits and 99.98% for misses.

Limitations. This technique relies on multiple threads and a mechanism for these to modify the same memory location. The easiest way of accomplishing this in the browser is with Web Workers and shared memory, which already implies the availability of known high-resolution timers [55, 57]. However, Architectural Reordering has an atypical behavioral footprint. Instead of continuously increasing a value and then querying it, which is signature behavior for a timing attack, the attacker thread performs two writes and a read to a single memory location per timing measurement, while the helper thread only performs a single write per measurement.

Convert + Measure: Architectural Reordering.
Eliminates explicit timing measurements by converting time differences into differences in an architecturally visible value.

6 CASE STUDIES

6.1 Extreme Amplification

6.1.1 Human Timers. As discussed in Section 4.3, our prefetch-based amplifier can produce comparatively huge timing differences based on a single initial cache hit or miss. It is worth asking whether the complete elimination of *all* sources of time from the execution environment would thwart CPU timing attacks at a fundamental level. To answer this question convincingly, we explore an artificially restrictive setting where there are no timers on the attacker's end, leaving them to rely only on their human perception.

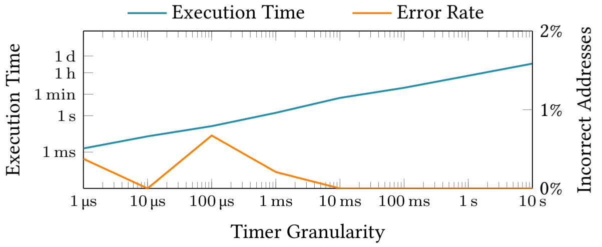


Figure 8: Constructing LLC eviction sets with the prefetch-based amplifier for extremely coarse-grained timers (200 runs for 1 µs-100 ms, 25 runs for 1 s, 1 run for 10 s).

We perform a study on fifteen human participants, aged 20-30. Each participant classifies 100 random single-shot side-channel observations as either fast or slow. The machine is an Intel Core i7-7700K, with which the participants interact over SSH. Participants are exposed to the measurement by a command line tool that prints `Start` (and `Stop`) when the traversal pattern starts (and ends), and are tasked to classify the runs corresponding to a single initial cache hit or cache miss. Based on some manual calibration, we parametrize the amplifier such that it produces a timing difference of 150 ms; the fast traversal takes 16 ms, whereas the slow traversal takes 166 ms. Like in other timing attacks, the participants first calibrate on a small number of practice observations (although, of course, the threshold is perceptual and not numerically quantified).

Together, the participants achieve an average accuracy of 98.4% (median 99%). Several participants achieve a perfect score. Note that the evaluation includes all error sources, such as human error, jitter due to SSH and I/O, and amplifier degradation due to noise.

6.1.2 Eviction Set Construction. As a relevant application, we also implement a routine to construct LLC eviction sets with arbitrarily coarse-grained timers. We do not rely on the availability of huge pages (2 MB or 1 GB), i.e., we only assume attacker control over the lower 12 bits of the physical address (4 KB pages).

We use the eviction set construction method due to Purnal et al. [51], together with the prefetch-optimization due to Guo et al. [24]. The routine tests individual cache lines for congruence in the LLC. To detect congruence, we use the prefetch amplifier. If addresses A and B are congruent, they constantly evict each other in a prefetch loop; otherwise, they do not. As A and B never enter the cache as anything other than the eviction candidate, amplifying this event is even more robust than for a generic side-channel setting (cf. Section 4.3). Lines that demonstrate congruence are accumulated in an eviction set until the desired number of addresses is obtained.

Figure 8 shows the execution time and error rate of the routine for varying timer precisions, on an Intel Core i7-7700K (16-way LLC). The error rate is the fraction of addresses that are not congruent with the randomly generated target. To emulate timers of arbitrary

```
1 victim_preamble();
2 x = calculate(_);          // <--- contention source ---+
3 load(f(x));                // load that depends on x    |
4 if (secret) {              //                           |
5     _ = calculate(_);      // <--- contention source ---+
6 }
```

Listing 3: Cross-core port contention leakage.

coarseness, we instrument calls to the rdtsc hardware counter. For the 1 s-granular timer, we use the Unix Epoch instead, i.e., the number of seconds elapsed since midnight on January 1, 1970.

We even attempt to construct an eviction set using a 10-second granular timer, representing an amplification of 8 orders of magnitude w.r.t. to the timing difference between a cache hit and a cache miss (e.g., 100 ns). With a runtime of less than 6 hours, the attempt is successful.

> **Amplify: Single-shot amplification up to seconds.**
> It is possible to amplify microarchitectural timing differences beyond any timer restriction that can be practically imposed.

6.2 Amplifying Stateless Leakage

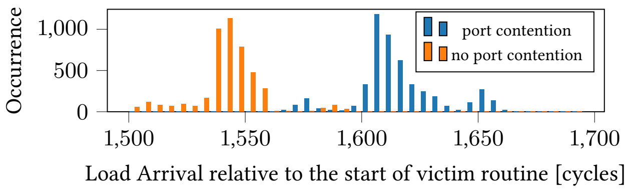


Figure 9: Fine-grained cross-core port contention attack.

6.2.1 Cross-Core Port Contention. Consider the code pattern in Listing 3, which leaks the boolean value of secret through port contention [4, 7, 54]. If the operations on lines 2 and 5 use the same execution ports, they interfere, delaying each other's execution. As ports are core-private resources, this stateless leakage is not directly visible to processes running on other cores. However, there is an implicit Convert performed by the victim code that still transmits this information; the presence or absence of contention introduces a secret-dependent delay on the load on line 3.

Fine-Grained Timer. We first expose the secret-dependent time of the memory access using a high-precision timer. Later, we apply ShowTime to decode the same information with a low-precision timer. We instantiate the contention sequence calculate() as 16 vsqrtpd (floating point square root) instructions. Figure 9 shows how the secret-dependent delay of the memory access, relative to the start of the victim program, can be picked up across cores by the high-precision Prime+Scope [51] attack. Note that the presence of the load itself does not encode any side-channel information, i.e., it happens independent of the secret. The time variation of the LLC eviction is roughly 70 cycles (i.e., less than 20 ns).

Coarse-Grained Timer. There are several challenges to exposing these fine-grained time variations to a low-precision timer. First, the Convert stage needs to implement an implicit threshold between the histograms in Figure 9, and the result of this threshold should be encoded in a stateful microarchitectural component from which it can be amplified. Second, the conversion requires a high timing sensitivity, comparable to accessing the scope line, which is already just sufficient enough to reveal the contention (cf. Figure 9).

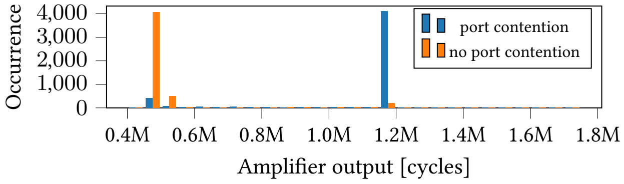


Figure 10: Coarse-grained cross-core port contention.

```
load(BCGAB);  // Reinstate A; makes pattern repeatable

// first leg                 // second leg
x = load(SCOPE);             y = fixed_delay();
                             y = load(G ^ y);

load(X ^ x ^ y);             // Evict A or E
```

Listing 4: Repeatable Time to Order conversion.

Our solution is the conversion pattern in Listing 4. The cache state is first prepared as follows. The LLC is prepared as in Prime+Scope, so the victim load will evict a designated cache line, i.e., the scope line. The L1 is prepared with the basic PLRU preparation pattern (cf. Section 4.1). The conversion is made repeatable with load(BCGAB), which evicts line X if it took the place of A (first leg won), but not if it took the place of E (second leg won). Therefore, if the monitored load evicts the scope line during any of the iterations of Listing 4, it is encoded in the L1 state. Before running the attack, we calibrate how often it needs to be repeated, relative to the start of the victim routine, to implicitly implement the threshold. As this conversion pattern takes only 30 cycles on average, it is precise enough to implement the necessary implicit threshold in Figure 9.

The complete ShowTime cascade is as follows. First, there is an unintentional conversion from port contention into an LLC eviction that occurs at a secret-dependent time. This secret-dependent time is converted into an *L1 Reordering* event, where the order depends on whether the LLC eviction occurs during the time that the attacker repeats the conversion pattern. Finally, the *L1 Reordering* event is amplified. Even though this cascade has several moving parts, the results are satisfying, as can be observed in Figure 10.

Discussion. Behnia et al. [5] exploit a code pattern similar to Listing 3. However, they question whether such a minor difference in load timing can be captured by a cache attack on the LLC. Therefore, they require all conversions to take place in the victim code, i.e., the victim itself should encode the time difference in the LLC replacement policy. In our work, we show that this requirement can be relaxed; high-precision LLC cache attacks can exfiltrate minute time differences directly, with and without fine-grained timers.

If an attacker can co-locate a process on the victim's CPU core, the contention may be exposed with a direct Time to Order conversion. We leave an exploration of this setting to future work.

6.2.2 Instantaneous CPU Frequency. Recently, attacks exploiting dynamic voltage and frequency scaling (DVFS) have been proposed [39, 70]. With DVFS, the instantaneous frequency of a CPU changes based on its power consumption which, in turn, may depend on the data being processed. We now explore whether information on the instantaneous CPU frequency can be exposed in the absence of direct interfaces (e.g., `cpufreq`) and fine-grained timers.

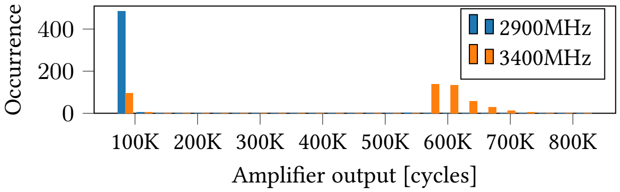


Figure 11: Exposing CPU frequency with crude timers.

Listing 5 (cf. Appendix A) contains the proof-of-concept code pattern. We observe that Time to Order races can be orchestrated to be sensitive to the instantaneous CPU frequency. We fix the CPU frequency using `sudo cpupower frequency-set` on an Intel Core i5-7500, running Rocky Linux 8.7. We set it to either 3400 MHz (the base frequency of the CPU) or 2900MHz, representing a 15% frequency adjustment. Figure 11 shows that the resulting histograms are clearly distinguishable. To our knowledge, we are the first to remark that the CPU frequency, an inherently stateless microarchitectural property, can be captured, converted, and amplified. We defer a comprehensive study of this phenomenon, as well as the achievable frequency granularity, to future work.

> **Convert + Amplify: Stateless Side Channels.**
> Stateless timing leaks can be exposed with coarse-grained timers.

6.3 ShowTime in Restricted Environments

With ShowTime, we can construct LLC eviction sets in JavaScript, which is a key prerequisite for several browser-based attacks, e.g., cross-core Prime+Probe [45], Rowhammer [13, 22], and Spook.js [2]. In addition, finding addresses that are L2/LLC-congruent permits the use of stronger PLRU amplifiers for the remainder of the attack.

With coarse-grained timers, the number of measurements replaces the number of memory references as the bottleneck for the execution time. Therefore, we use the group elimination method by Vila et al. [67] rather than the Prime+Scope method [51]. We modify Vila's JavaScript code [65] to use ShowTime as the measurement. In particular, we use Time to Order to translate the LLC eviction signal to the L1 cache and use the distance-3 PLRU amplifier for robustness (cf. Listing 6 in Appendix A for details.)

We start with an initial set that is a superset of an eviction set with 95% probability (cf. [67]) and exclude the runs where this is not the case. To obtain the ground truth, we verify the correctness of the eviction set using `/proc/pagemap` [65]. On a non-isolated website in Chrome 108, with a `performance.now()` precision of 100 µs, we construct a fully correct eviction set in 176 out of 250 runs, with a median runtime of 25 seconds. For this proof-of-concept implementation, we did not exhaust all possible optimization opportunities.

Table 5: Converting CPU Side Channels.

Source	Method	Input Channel	Output Channel
Behnia et al. [5]	OoO Execution	Port/MSHR Cont.	LLC
Aimoniotis et al. [3]	ROB Size	ROB Pressure	L1/LLC
Wang et al. [70]	DVFS	Power	Freq. / Time
Back-Invalidation	**LLC Inclusiveness**	**LLC**	**L1**
Time to Order	**OoO Execution**	**Any**	**L1/LLC**
Arch. Reordering	**Thread Parallelism**	**Any**	**Data**

As the objective of this work is to study microarchitectural attacks in the face of coarse-grained timers, we also made no attempts to increase the effective precision of the timing sources themselves.

> **Convert + Amplify: ShowTime in the browser.**
> ShowTime applies to restricted browser settings. It can be used to construct LLC eviction sets with coarse-grained timers.

7 RELATED WORK

Multi-Shot Amplification. Mcilroy et al. [42] provide a theoretical argument for the availability of arbitrary multi-shot timing amplification on processors implementing optimizations. Wikner et al. [73] and Schwarzl et al. [59] consider multi-shot amplification for mounting Spectre attacks from JavaScript. Some other works cope with low-resolution timers by aggregating the latency of many memory accesses [60, 61], with the drawback of losing all spatial information of the side channel. Rokicki et al. [54] amplify (stateless) port contention from JavaScript in a covert channel setting, where multi-shot measurements are possible.

Multi-shot amplification is also used for the software-based exploitation of physical side-channels (e.g., power consumption [37] and CPU frequency [39, 70]). Future work should investigate the feasibility of single-shot amplification for these attack vectors.

Existing Conversions. Table 5 summarizes the prior work and our contributions in the space of converting CPU side channels. Behnia et al. [5] convert several sources of core-private contention to LLC caching status by exploiting specifics of the LLC replacement policy. Aimoniotis et al. [3] exploit that incorrectly speculated loads only get executed if they fit in the reorder buffer (ROB) [74], converting ROB contention into caching status. Our work contributes simple conversions of several side channels into state differences that are amenable to single-shot amplification.

Disabling Timing Sources. Browsers already cripple timers [16, 17, 44, 63, 68, 72], but this is also proposed for native (mobile/desktop/cloud) code [41, 64]. Prior work demonstrates that, in some cases, attackers can build [19, 55, 57, 58] or simply bring [52] their own timing sources. However, our work practically demonstrates that even when attackers cannot use these methods, restricted timers are not a holistic countermeasure against timing attacks.

Note that our findings do not threaten the validity of other side-channel countermeasure classes, such as constant-time programming [32], isolation [11, 48] or randomization [71].

Concurrent Work. In concurrent work, Xiao et al. [75] leverage out-of-order execution ("race gadgets") to convert microarchitectural state changes, similar to one of our conversion routines (Time to Order, cf. Section 5.2). However, they do not consider amplifying stateless channels. They also contribute single-shot amplifiers ("magnification gadgets"), including the *L1 Reordering* PLRU amplifier, and others that are not cache-based. Though they suggest that arbitrary amplification can be achieved, they do not demonstrate amplifying timing differences beyond 100 μs. In our experiments, we overcome several practical challenges to obtain timing differences that are larger by one to four orders of magnitude.

Another concurrent work [30] uses transient execution to encode the caching status of one cache line into many cache lines. In this manner, they obtain single-shot amplification of cross-core cache events. Similar to our work, they also construct LLC eviction sets in a browser environment using a 100 μs timer.

8 CONCLUSION

In this paper, we contributed the ShowTime framework to expose arbitrary microarchitectural timing leaks in a single shot to coarse-grained timers. Our techniques can capture cross-core and stateless microarchitectural leaks, bypass currently imposed timer restrictions, and even amplify nanosecond-range timing differences such that they are detectable by humans.

ACKNOWLEDGMENTS

We thank the anonymous AsiaCCS reviewers for their feedback and the humans for participating in the timer study. This research is partially funded by the European Research Council (ERC #101020005) and the Flemish Government through the FWO project TRAPS. It was also supported by the CyberSecurity Research Flanders (#VR20192203), Horizon Europe (#101070008) and the Research Fund KU Leuven. Antoon Purnal is supported by a grant of the Research Foundation - Flanders (FWO).

REFERENCES

[1] Andreas Abel and Jan Reineke. 2020. nanoBench: a Low-overhead Tool for Running Microbenchmarks on x86 Systems. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[2] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. 2022. Spook.js: Attacking chrome strict site isolation via speculative execution. In *IEEE Symposium on Security and Privacy (S&P)*.

[3] Pavlos Aimoniotis, Christos Sakalis, Magnus Själander, and Stefanos Kaxiras. 2021. Reorder Buffer Contention: A Forward Speculative Interference Attack for Speculation Invariant Instructions. In *IEEE Computer Architecture Letters*.

[4] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port contention for fun and profit. In *IEEE Symposium on Security and Privacy (S&P)*.

[5] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa Alameldeen. 2021. Speculative Interference Attacks: Breaking Invisible Speculation Schemes. In *ASPLOS*.

[6] Daniel J Bernstein. 2005. Cache-timing attacks on AES.

[7] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[8] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *IEEE Symposium on Security and Privacy (S&P)*.

[9] Samira Briongos, Ida Bruhns, Pedro Malagón, Thomas Eisenbarth, and José M. Moya. 2021. Aim, Wait, Shoot: How the CACHESNIPER Technique Improves Unprivileged Cache Attacks. In *IEEE European Symposium on Security and Privacy (EuroS&P)*.

[10] Samira Briongos, Pedro Malagon, Jose M. Moya, and Thomas Eisenbarth. 2020. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *USENIX Security Symposium*.

[11] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium*.

[12] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. 2022. Don't Mesh Around: Side-Channel Attacks and Mitigations on Mesh Interconnects. In *USENIX Security Symposium*.

[13] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. 2021. SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript. In *USENIX Security Symposium*.

[14] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *USENIX Security Symposium*.

[15] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2018. BranchScope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices* (2018).

[16] Google. 2020. Making your website "cross-origin isolated" using COOP and COEP. https://web.dev/coop-coep/.

[17] Google. 2021. Align performance API timer resolution to cross-origin isolated capability - Chrome Platform Status. https://chromestatus.com/feature/6497206758539264.

[18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*.

[19] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *Network and Distributed System Security Symposium (NDSS)*.

[20] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-channel Protection Using Hardware Transactional Memory. In *USENIX Security Symposium*.

[21] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[22] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[23] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security Symposium*.

[24] Yanan Guo, Xin Xin, Youtao Zhang, and Jun Yang. 2022. Leaky Way: A Conflict-Based Cache Covert Channel Bypassing Set Associativity. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[25] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy (S&P)*.

[26] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *Cryptographic Hardware and Embedded Systems (CHES)*.

[27] Intel. 2019. Intel Transactional Synchronization Extensions (Intel TSX) Asynchronous Abort. https://software.intel.com/security-software-guidance/deep-dives/deep-dive-intel-transactional-synchronization-extensions-intel-tsx-asynchronous-abort.

[28] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing – and Its Application to AES. In *IEEE Symposium on Security and Privacy (S&P)*.

[29] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[30] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. 2023. The Gates of Time: Improving Cache Attacks with Transient Execution. In *USENIX Security Symposium*.

[31] 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 *IEEE Symposium on Security and Privacy (S&P)*.

[32] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO*.

[33] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. 2022. Half-Double: Hammering From the Next Row Over. In *USENIX Security Symposium*.

[34] Andreas Kogler, Daniel Weber, Martin Haubenwallner, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. Finding and Exploiting CPU Features using MSR Templating. In *IEEE Symposium on Security and Privacy (S&P)*.

[35] David Kohlbrenner and Hovav Shacham. 2016. Trusted Browsers for Uncertain Times. In *USENIX Security Symposium*.

[36] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*.

[37] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-based Power

Side-Channel Attacks on x86. In *IEEE Symposium on Security and Privacy (S&P)*.

[38] 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 *USENIX Security Symposium*.

[39] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. 2022. Frequency throttling side-channel attack. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[40] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks Are Practical. In *IEEE Symposium on Security and Privacy (S&P)*.

[41] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. Timewarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *International Symposium on Computer Architecture (ISCA)*.

[42] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv:1902.05178* (2019).

[43] MDN. 2020. Firefox 79 release notes for developers. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Releases/79#javascript.

[44] MDN. 2022. performance.now() - Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/Performance/now.

[45] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[46] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*.

[47] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. 2021. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *USENIX Security Symposium*.

[48] Dan Page. 2005. Partitioned cache architecture as a side-channel defence mechanism. In *IACR Cryptol. ePrint Arch. 2005/280*.

[49] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-cpu Attacks. In *USENIX Security Symposium*.

[50] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. 2021. Systematic Analysis of Randomization-based Protected Cache Architectures. In *IEEE Symposium on Security and Privacy (S&P)*.

[51] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. 2021. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[52] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. 2022. Double Trouble: Combined Heterogeneous Attacks on Non-inclusive Cache Hierarchies. In *USENIX Security Symposium*.

[53] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[54] Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. 2022. Port Contention Goes Portable: Port Contention Side Channels in Web Browsers. In *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*.

[55] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. 2021. Sok: In Search of Lost Time: A Review of JavaScript Timers in Browsers. In *IEEE European Symposium on Security and Privacy (EuroS&P)*.

[56] Stephen Röttger and Artur Janc. 2021. A Spectre proof-of-concept for a Spectre-proof web. https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html.

[57] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *Financial Cryptography and Data Security*.

[58] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[59] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Michael Schwarz, and Daniel Gruss. 2022. Robust and Scalable Process Isolation Against Spectre in the Cloud. In *European Symposium on Computer Security (ESORICS)*.

[60] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. 2021. Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. In *USENIX Security Symposium*.

[61] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *USENIX Security Symposium*.

[62] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. 2022. SecSMT: Securing SMT processors against contention-based covert channels. In *USENIX Security Symposium*.

[63] The Tor Project. 2015. Commit: Bug 1517: Reduce precision of time for Javascript. . https://gitlab.torproject.org/tpo/applications/tor-browser/-/commit/dcd5fcc102a3eb19c20013542fa3ca399db66da4.

[64] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. 2011. Eliminating Fine Grained Timers in Xen. In *ACM Workshop on Cloud Computing Security (CCSW)*.

[65] Pepe Vila. 2019. Tool for testing and finding minimal eviction sets. https://github.com/cgvwzq/evsets.

[66] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. 2020. CacheQuery: Learning replacement policies from hardware caches. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[67] Pepe Vila, Boris Köpf, and José F. Morales. 2019. Theory and Practice of Finding Eviction Sets. In *IEEE Symposium on Security and Privacy (S&P)*.

[68] W3C. 2022. High Resolution Time. https://www.w3.org/TR/hr-time-3/.

[69] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. 2022. MeshUp: Stateless cache side-channel attack on CPU mesh. In *IEEE Symposium on Security and Privacy (S&P)*.

[70] 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 *USENIX Security Symposium*.

[71] Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *International Symposium on Computer Architecture (ISCA)*.

[72] WebKit. 2018. What Spectre and Meltdown Mean For WebKit. https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/.

[73] Johannes Wikner, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2022. Spring: Spectre Returning in the Browser with Speculative Load Queuing and Deep Stacks. In *Workshop On Offensive Technologies (WOOT)*.

[74] Henry Wong. 2013. Measuring Reorder Buffer Capacity. https://blog.stuffedcow.net/2013/05/measuring-rob-capacity/

[75] Haocheng Xiao and Sam Ainsworth. 2023. Hacky Racers: Exploiting Instruction-Level Parallelism to Generate Stealthy Fine-Grained Timers. In *ASPLOS*.

[76] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *IEEE Symposium on Security and Privacy (S&P)*.

[77] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security Symposium*.

[78] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a Timing Attack on OpenSSL Constant-time RSA. *Journal of Cryptographic Engineering* (2017).

[79] Zirui Neil Zhao, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. 2022. Binoculars: Contention-Based Side-Channel Attacks Exploiting the Page Walker. In *USENIX Security Symposium*.

Appendix

A EXTRA CONVERSION PATTERNS

```
// Prepare prefetch amplifier
dep = prefetchNTA(A);

// Shared Dependency
dep = load(X + dep);   // Cache miss

// Compute chain races against memory loads,
//  the outcome of this race is frequency-dependent

// First leg              // Second leg
dep1 = load(Y + dep);     dep2 = COMPUTE_CHAIN(dep);
dep1 = load(A + dep1);    dep2 = load(B + dep2);

// evicts A if the first leg won
dep = load(C + dep1 + dep2);

// Amplify time difference
prefetch_amplifier(D, A); // Fast if A is cached
```

Listing 5: Instantaneous frequency measurement. The loads of X and Y are cache misses, and only A-D map to the same LLC set.

```
// Test whether group still evicts victim
dep = load(victim);
dep = evict(candidate_set ^ dep);

// Start timer
start = performance.now();

// Prepare PLRU amplifier
dep = prepare_PLRU(dep);

// first leg                // second leg
dep1 = load(victim ^ dep);  dep2 = delay(dep);
dep1 = load(D ^ dep1);      dep2 = load(H ^ dep2);

// amplify the difference
amplify_L1(dep1, dep2);

// End timer
end = performance.now();
```

Listing 6: Constructing LLC eviction sets.