Hacky Racers: Exploiting Instruction-Level Parallelism to Generate Stealthy Fine-Grained Timers

Haocheng Xiao
University of Edinburgh
Edinburgh, UK
haocheng.xiao@ed.ac.uk

Sam Ainsworth
University of Edinburgh
Edinburgh, UK
sam.ainsworth@ed.ac.uk

ABSTRACT
Side-channel attacks pose serious threats to many security models, especially sandbox-based browsers. While transient-execution side channels in out-of-order processors have previously been blamed for vulnerabilities such as Spectre and Meltdown, we show that in fact, the capability of out-of-order execution itself to cause mayhem is far more general.

We develop Hacky Racers, a new type of timing gadget that uses instruction-level parallelism, another key feature of out-of-order execution, to measure arbitrary fine-grained timing differences, even in the presence of highly restricted JavaScript sandbox environments. While such environments try to mitigate timing side channels by reducing timer precision and removing language features such as SharedArrayBuffer that can be used to indirectly generate timers via thread-level parallelism, no such restrictions can be designed to limit Hacky Racers. We also design versions of Hacky Racers that require no misspeculation whatsoever, demonstrating that transient execution is not the only threat to security from modern microarchitectural performance optimization.

We use Hacky Racers to construct novel backwards-in-time Spectre gadgets, which break many hardware countermeasures in the literature by leaking secrets before misspeculation is discovered. We also use them to generate the first known last-level cache eviction set generator in JavaScript that does not require SharedArrayBuffer support.

CCS CONCEPTS
• Security and privacy → Web application security; Side-channel analysis and countermeasures; • Computer systems organization → Superscalar architectures.

KEYWORDS
Microarchitectural Security, Caches, Spectre, Instruction-level Parallelism, JavaScript

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:
Haocheng Xiao and Sam Ainsworth. 2023. Hacky Racers: Exploiting Instruction-Level Parallelism to Generate Stealthy Fine-Grained Timers. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS ’23), March 25–29, 2023, Vancouver, BC, Canada. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3575693.3575700

1 INTRODUCTION
The disclosures of Spectre [37] and Meltdown [42] in 2018 have moved side channels into the mainstream. Various performance-optimization techniques in today’s processors, such as transient execution, and caches [7–9, 15, 16, 19, 28, 30, 35, 37, 39, 40, 42, 44, 53, 54, 58, 63, 68, 70, 74, 77–79, 81, 88, 90, 91], have been shown to be exploitable by attackers to leak or exfiltrate information. This has particularly manifested in browser security, where large amounts of untrusted sandboxed JavaScript code is executed. Although JavaScript is sandboxed, it is still vulnerable to most architectural attacks, e.g., Spectre [7, 37], Rowhammer [29], prime+probe [48] and many others [27, 32, 52, 73, 76, 80]. Still, timing side-channel attacks rely on being able to reliably measure timing differences, and so the precision of the timers provided by browsers’ APIs has been decreasing [6, 47]. Although the level of assurance this mitigation actually gives has been hotly debated [5, 38, 45, 72], vendors still tend to resort to this whenever other more effective mitigations, such as patching processor microarchitecture, disabling hardware optimization techniques, or process-level isolation [57] are unavailable and/or degrade performance to too high a degree [34, 64, 66, 84, 89, 94].

A prime example of this is that when SharedArrayBuffer, a JavaScript multi-threaded language feature, was found to provide an indirect fine-grained timer through counters and thread-level parallelism [69], it was temporarily removed from mainstream browsers [21] as a response to Spectre [37]. It is still blocked in recent mitigation proposals such as Chrome Zero [67], in the Tor Browser [1], and in Firefox and Chrome for websites that have not opted in to cross-site isolation [13].

In this paper we break the illusion of security brought by this timer-coarsening mitigation, and by the removal of SharedArrayBuffer. We develop Hacky Racers, a new class of timing gadgets that can be used to measure arbitrary fine-grained timing differences on out-of-order processors without any previously removed or potentially removable language features [21], cross-thread contention [69], or indeed anything other than simple arithmetic operations, branches, loads, and coarse-grained timers.

While attacks such as Spectre [37] depend on the transient-execution capability of an out-of-order processor, and SharedArrayBuffer timers on thread-level parallelism that can be disabled, we instead attack instruction-level parallelism (ILP): the ability for two or more data-independent instruction sequences from the same
thread to execute simultaneously, creating a race for which executes first. The key insight is that an attacker can use ILP to generate sequences of instructions that can be comparatively timed against each other, even in the absence of any direct source of time. The actual execution order between two racing instructions can be converted into long-lasting cache-state changes, such as an L1 eviction or reordering two cache fills, by racing (section 5) two different sections of independent code against each other. This can then be magnified (section 6) by repeatedly sampling the cache [62] or causing contention in the pipeline, to convert the fine-grained timing difference into a coarse-grained timing difference, undiminished by low resolution or noise.

This paper makes the following contributions:

1. We propose a new method of exploiting out-of-order execution, Hacky Racers, which use instruction-level parallelism to generate fine-grained timers.
2. We introduce racing gadgets, used to differentially time one event relative to another, and leave a state accordingly as an input for magnifier gadgets.
3. We introduce magnifier gadgets, to amplify the timing difference between different micro-architecture states caused by a small difference, such as a speculative memory access, or the order of two cache fills.
4. We prove the efficacy of Hacky Racers through implementing several attacks in a browser without using SharedArrayBuffer, including a new variant of Spectre V1 [37], SpectreBack, that can leak secrets backwards-in-time, to before any misspeculation is discovered.
5. We demonstrate that Hacky Racers can resurrect side-channel attacks thought to have been purged via timer coarsening, by implementing the first known eviction-set generator in JavaScript without SharedArrayBuffer.

To be clear, Hacky Racers are not as big a threat as transient-execution attacks such as Spectre [37], as while transient-execution attacks directly leak secrets, Hacky Racers (and instruction-level parallelism in general) instead leak time. Still, that means that Hacky Racers can form the critical part in making information-leakage attacks (such as Spectre) feasible, by making information-recovery of them practical even in extremely restricted environments. We have disclosed our findings to the Tor Browser, Chrome, Firefox, and Cloudflare. The Hacky-Racers attack code is open-sourced at https://github.com/FxPiGaAo/Hacky-Racer.

2 BACKGROUND AND RELATED WORK

Here we first introduce how timers are used to facilitate side-channel attacks. We then discuss current methods of timing in browsers and their mitigations, followed by background on the out-of-order execution of modern processors that we use to generate the instruction races used in Hacky Racers. Finally, we discuss Spectre attacks.

2.1 Timing in JavaScript Attacks

Timers are typically used in JavaScript attacks in two places: in the receiving stage of a side channel (e.g. to time the presence/absence of a cache line [7] or execution-unit contention [25]), in the last-level cache (LLC) eviction set (EV) profile stage [22, 29, 48, 52, 53, 75, 83, 92] (to allow efficient preparation of the cache before an attack), or both.

The precision of the timer matters when the attacker needs to time the access latency to one or multiple addresses to distinguish whether a cache miss exists or not. Last-level-cache (LLC)-based channels [30, 48, 67], such as Flush+Reload [93], Prime+Probe [43, 49, 51, 72], Evict+Time [49], Evict+Reload [30], Reload+Refresh [18, 88] and Evict+Prefetch [28], are prevalent, since they are still effective even if the attacker and victim never share the same core. Though the timing of an LLC miss, which is used in the eviction-set profiling and the secret transmission stage, is more coarse grained than an L1 cache miss, its timing difference is still around 100ns, and so cannot be timed natively with the coarse-grained timers (section 2.2) in today’s browsers.

A precise timer is critical to some other (non-timing-side-channel) attacks, such as rowhammer.js [29] and Spook.js [7], as they also require LLC eviction-set profiling to actively trigger victim cache misses from the attacker’s side. Finally, there are other side-channel attacks that require timing information such as website fingerprinting [48, 59, 72, 73], inferring neural-network architectures [90] and breaking AES [32].

2.2 Timers in Browsers and Mitigation

Sources of time in browsers can be generated intentionally by APIs, or unintentionally by other system behavior [60, 69].

Performance.now() is the most precise timer provided natively by browsers’ APIs. Its precision has changed over time as a response to attacks. For example, its resolution was decreased to 5 μs in both Chrome and Firefox in 2015 as the first browser side-channel attack [48] emerged. In early 2018, Spectre [37] forced vendors to further decrease its precision to 2 μs in Firefox 59, and 100 ms in Chrome with a 100 ms jitter [60]. After site isolation [57], a mitigation to remove secrets from the address space, was added in Chrome and Firefox, the precision restriction was loosened to 5 μs plus jitter in Chrome [6] and 5 μs in Firefox [46] within the same origin.

SharedArrayBuffer was introduced by ECMAScript 2017 [31] to facilitate cross-thread communication. It provides shared memory between multiple workers, enabling simultaneous data reading and writing between threads. Based on this feature, Schwarz et. al. [69] built a high-resolution timer with precision around 2 · 15 nanoseconds. A shared value among a main thread and a subthread is allocated in this SharedArrayBuffer. The subthread serves as a counting thread, incrementing the value continuously in an infinite loop. The main thread then gets timestamps by reading the value.

Similar to the high-resolution performance.now() API, the SharedArrayBuffer feature was also disabled [2] as a response to Spectre [37] and re-enabled after the implementation of site isolation in Chrome and Firefox only for cross-origin isolated pages [13]. It is still disabled in Tor [3] and Chrome Zero [67].

Others Apart from lowering precision, other countermeasures have been proposed, such as fuzzy timers [38, 82] and limiting shared memory and message passing. In fuzzy time, the clock edge

---

1Resolution can also be customized to a larger value by enabling privacy.resistFingerprinting [47] in Firefox, and 100 μs (from the default 5 μs) in non-isolated contexts in Chrome.
is randomly perturbed, further reducing the final observed precision. This was thought to mitigate the edge-thresholding technique [69], which otherwise has the potential to recover the resolution by adding extra instructions to reach the threshold.

2.3 Repetition and Magnifier Gadgets

To observe fine-grained timing difference with a coarse-grained timer, the signal must in some way be amplified. Here we divide such methods into two categories: repetition gadgets, which repeat the attack many times, and magnifier gadgets, which require the attack to be performed only once.

Repetition gadgets repeat the whole extract-then-transmit process to accumulate the timing difference from each iteration. For example, Skarlatos et al. [74] denoise their SGX attack by repeatedly triggering replay on illegal instructions without actually incurring a page fault. McIlroy et al. [45] propose a gadget to create timing difference for Spectre gadgets [37] that can be observed by the coarse-grained timer in a browser, by repeatedly triggering the attack. Schwarz et al. [71] implement a similar attack on Cloudflare Workers [4], the serverless execution framework, where timing can only be achieved by a remote timing server. However, repetition on its own has the risk of cancelling out the timing difference between different stages (section 7.1).

In contrast, a magnifier gadget first extracts and transmits the secret into state differences once, and then executes instructions that both maintain the state difference and accumulate the timing difference at the same time. Therefore, this gadget will not suffer from noise from stages such as branch mistraining. We re-purpose the magnifier gadget introduced by the leaky.page Spectre attack [62], which attacks PLRU caches, to time arbitrary sequences of instructions, and introduce new magnifier gadgets that avoid any transient execution, and work for other cache replacement strategies (section 6).

Orthogonally, attackers can also leave multiple state differences [86] after the secret is extracted once, which can also increase the timing difference by a constant (typically limited) multiple – though further timer coarsening by a similar constant can mitigate the effects.

2.4 Out-of-Order Execution

Out-of-order execution is an almost ubiquitous [37] processor microarchitectural optimization technique, designed to exploit instruction-level parallelism, by executing many independent instructions from the same thread simultaneously when there are no data dependencies between them. This optimization must obey the programmer’s sequential view of execution, and so instructions commit in-order at the backend of the pipeline. Still, the true out-of-order nature of execution may still leave traces in structures outside the programmer’s model, such as the cache, which we can pick up to infer which executed first.

In this paper, we define a path (section 4) as an instruction sequence that currently does not have any external data dependence, and could thus be processed independently of any instructions outside of that sequence. This independence allows execution to flow within separate paths simultaneously.

2.5 Speculative Execution Attacks

To find useful work, an out-of-order processor must speculate on the future flow of instructions, such as via branch predictors. This incorrect execution will be rolled back, but may still leave traces that can be found by an attacker.

Misspeculated (or transient) execution was famously attacked by Spectre [37]: the branch predictor or branch target buffer can be poisoned to cause incorrect instructions (such as out-of-bounds accesses to arbitrary parts of the address space) to execute speculatively, and leave traces in the cache. Meltdown [42] similarly attacked the behavior of some Intel cores to incorrectly forward on the values of exceptional loads to other transient instructions.

Hacky Racers are different. Though transient execution makes the construction of its gadgets simpler, the attack requires no transient execution at all. The correct execution of instructions, executed in the wrong order, is still a threat to browser security. Hacky Racers also provide new perspectives on transient execution attacks. In section 7.3, we will show new racing gadgets that allow leaking of data in ways that many proposed mitigations cannot eliminate. Many other speculative execution attacks based on contention, of either arithmetic logic units (ALUs) in the core [25] or miss status holding registers (MSHRs) in the cache [15], could also be validly formulated as Hacky Racers racing gadgets.

3 THREAT MODEL

We assume an arbitrary side-channel attack in a JavaScript application, such as an advertisement in a web browser trying to access secrets outside its sandbox, that needs to time one or several operations in order to succeed, such as the probe operation in a prime+probe [51]. We further assume that the timing difference brought about by this side channel is fine-grained – of the order of 100 nanoseconds or smaller, thus cannot be observed by current browsers’ JavaScript native timers. We also assume the attacker can execute any valid JavaScript code within their sandbox, but that any timers or other forms of time (section 2.2), are limited to 5μs granularity. The majority of our attacks also work for as coarse a granularity as has ever been introduced in a browser (100ms) and higher, but we use 5μs as the threshold for success of an attack.

We do not require the attacker to have access to any cross-thread primitives. The attacker will succeed if they are able to measure this difference and thus construct an information channel. This gives Hacky Racers the potential to resurrect or accelerate attacks that were previously hindered by timing obfuscation [38, 67].

We assume the attacker is executing on a processor with out-of-order execution, as is typical on modern devices [37], able to execute tens-to-hundreds of instructions into the future. The instruction length of operations to be measured is limited by the reorder buffer of a processor in practice. However, we find that very few concurrently executing instructions are required for success.

---

2 Concurrent to this work, Rokicki et al. [61] also exploited ILP to build a port-contention channel in JavaScript that can be used to determine a user’s CPU model. Their attack can be used to replace the simultaneous multi-threading (SMT) feature relied by the original attack with ILP contention, but still uses a SharedArrayBuffer based timer, which Hacky Racers can replace.
We then demonstrate how to embed the expression whose timing we would like to observe, named the target expression or \( G \). Where a fine-grained timer is unavailable, such as Cloudflare Workers [4] and, Apple M1 processors and other ARM systems [41, 56]. We leave these to future work for simplicity.

4 PATH CONSTRUCTION

Here we define paths as a unit of simultaneous execution within a single thread (exemplified in fig. 1) on an out-of-order processor. Between two independent paths, \( p_a \) and \( p_b \), there must be no data dependencies, and thus they may execute entirely in parallel with each other on an out-of-order processor. Likewise, no instruction within the path should be data dependent on any instruction outside the path, except for instructions that come before the entire path in program order. Within a path, there may be one or more chains which are connected via data dependence, such that no two instructions within each chain can execute simultaneously or out-of-order. Note that paths are defined in order to simplify the synchronization and concurrency requirements of Hacky Racers.

We first introduce the construction process of paths that satisfy the synchronization and concurrency requirements of Hacky Racers. We then demonstrate how to embed the expression whose timing we would like to observe, named the target expression or \( \text{Expr} \), into a path as the first step of our Hacky Racer.

Ultimately, this path will compete against another path, the baseline path, as part of the racing gadgets of section 5, which transform the race result of the two paths into a state change in the system. This state change can then be input into magnifier gadgets to amplify the side channel to something observable via coarse-grained timer, even after only a single attack instance.

4.1 Path Synchronization

Code listing 1 shows an example of a construction of two independent paths, \( p_a \) and \( p_b \), that are eligible for execution simultaneously. In this example, each path consists of a single chain. To synchronize their starting execution, we ensure the first instruction in the chains in both paths have a common data dependency, in instruction \( A \).\(^3\). We ensure that instruction A, loading Array[0], will incur a cache miss that delays executions of both paths to avoid frontend fetch/decode contention, meaning all instructions will have reached the out-of-order backend by the time any is eligible for execution. As a result, \( p_a \) and \( p_b \) start at the same time and run in parallel.

4.2 Expression Embedding

In addition to synchronizing the start of each path’s execution, to observe the execution timing of the target expression, it should be embedded into a single path, named the measurement path. The path may also be required to end with an attacker-defined result as its terminator instruction, transitively dependent on the final instruction of every chain within the path. In practice this terminator instruction will serve either as a branch-condition variable or as a data-access index, to be used as a side channel to infer their relative timing, as will be introduced in section 5. To satisfy these two restrictions, we construct both a pre-extension and post-extension around every chain within the original target expression. The pre-extension’s output encodes the input data for the target expression, ensuring that all data inputs to the function depend on a single head instruction at the start of the path, while the post-extension converts the original outputs of the target expression into inputs to an attacker-controlled instruction, to ensure that this instruction only executes following the completed execution of all outputs.

In our example, \( \text{myFunction}() \) is shown both in the red-dotted box of figure 2 and in the OriginFunc() of code listing 2. The transformed function contains the measurement path transformed from the target expression. Between the pre-extension and target expression, the indices used to load A and C in the OriginFunc() are replaced by F and G, which encode each index as a data dependency of E. As is discussed in section 4.1, the load of E will suffer a cache miss to synchronize the starts of the two paths. In the post-extension, var H is created to depend on both B and D, ensuring the completion of the whole path only after B and D have finished execution. Finally, in var I, an offset is added to H to achieve the desired extension.

Thus, the target operation is embedded into a path that produces a particular output only once the full operation has completed. In the next section, we will describe how to synchronize it against a

---

\(^3\) Where paths have multiple chains within them as in figure 2, the first instruction in every chain must depend (transitively) on this common dependency.
5 RACING GADGETS

We develop racing gadgets to measure timing relatively with reference to a path with a known constant execution time, which we call a baseline path, path\(_b\). The target expression to be measured, Expr\(_t\), is placed in the measurement path, path\(_m\), starting at the same time as the baseline path. Thus, the path with a shorter execution time will finish first. This completion order between these two paths leaves a micro-architectural difference that will serve as the input of the magnifier gadget.

5.1 Transient Presence/Absence (P/A) Racing Gadget

This gadget converts the completion order into whether there is a transient cache access to a specific address or not. This is achieved by altering how quickly a mispredicted branch is corrected, depending on the operation time that we are interested in. The cache-state change caused by the transient access serves as input to the corresponding magnifier gadget.

Assume \( T \) is threshold which we would like to distinguish whether the execution time of Expr\(_t\) is larger or smaller than. We prepare two operation sequences path\(_m\)(Expr\(_t\), \( x \)) and path\(_b\)(\( x \))^4. The variable \( x \) controls the final data output of path\(_m\). path\(_m\)(Expr\(_t\), \( x \)) satisfies the following requirements:

\[
\begin{align*}
& a) \text{path}_m(\text{Expr}_t, 0) = 1 \text{ and path}_m(\text{Expr}_t, 1) = 0. \\
& b) \text{previous execution of path}_m(\text{Expr}_t, 0) \text{ will not affect the execution time of path}_m(\text{Expr}_t, 1). \\
& c) \text{Time(Expr}_t\text{)}_\text{low} < \; T \; < \; \text{Time(Expr}_t\text{)}_\text{high} \Rightarrow \text{Time(path}_m(\text{Expr}_t, 1)\text{)}_\text{low} < \; T' \; < \; \text{Time(path}_m(\text{Expr}_t, 1)\text{)}_\text{high}
\end{align*}
\]

Requirements a) and b) are needed due to the training phase, since its output serves as a condition variable to trick the branch predictor into executing transient code. We set \( x = 0 \) during the training phase and \( x = 1 \) during the detection phase.

\[
\begin{align*}
& d) \text{No instruction in path}_b(\cdot) \text{ can have a data dependency on any instruction in path}_m(\text{Expr}_t, 1), \text{ and vice versa} \\
& e) \text{path}_b(\cdot) \text{ and path}_m(\text{Expr}_t, 1) \text{ should be started at almost the same time.}
\end{align*}
\]

Requirement d) is implied by the definition of path, and is necessary to allow the two paths to execute independently (and thus race against each other), while requirement e), to make the race between the two fair (and thus increase precision), is satisfied by

---

\(^4\text{The construction to satisfy these restrictions is discussed in section 4.}\)
letting the first instruction in both paths to be dependent on the same cache miss (section 4.1).

Finally we construct the timing gadget as follows (indicates that access[A] has a data dependence on the result of path_b(), thus it will be executed after all of the instructions in path_b() have executed even on an out-of-order processor):

\[
\text{if}(\text{path}_m(x)) \\
\quad \text{path}_b() \rightarrow \text{access}[A];
\]

Before this gadget is executed, we train the branch predictor through executing this code snippet with \(x = 0\). When we execute it with \(x = 1\), the mispredicted path_b() will be executed in parallel with path_m() until one finishes. If \(\text{Time(Expr)} > T\), the memory request of access[A] will be sent after path_b() finishes before path_m(). Otherwise when path_m() finishes first, it will roll back before it reaches access[A].

5.2 Non-Transient Reorder Racing Gadget

Here we introduce another racing gadget that does not rely on transient execution, unlike Spectre [37]. No misspeculation is required for this gadget, so it is unaffected by hardware mitigation schemes for Spectre [10, 89, 95] and demonstrates that we are attacking a fundamentally different property of out-of-order execution. Here, inspired by a gadget used as part of speculative interference attacks [15] to defeat invisible transient caching mechanisms for Spectre resistance [89], we convert completion order into the relative order of two memory accesses, A and B. We construct the gadget as follows:

\[
\text{path}_m() \rightarrow \text{access}[A]; \\
\text{path}_b() \rightarrow \text{access}[B];
\]

And with the following requirements:

a) \(\text{Time(Expr)}_{\text{low}} < T < \text{Time(Expr)}_{\text{high}} \Rightarrow \text{Time(path}_m(\text{Expr}))_{\text{low}} < T' < \text{Time(path}_m(\text{Expr}))_{\text{high}}\)

b) No instruction in path_b() can have a data dependency on any instruction in path_m(), and vice versa

c) path_b() and path_m() should be started at the same time.

These are similar to those of the transient P/A racing gadget except that the first two restrictions are removed, as this gadget does not include branches. Because path_b() and path_m() start execution simultaneously, the one which completes first will issue the subsequent memory access earlier than the other.

6 MAGNIFIER GADGETS

The racing gadgets of Section 5 place the system in one of two different micro-architectural states: one to transmit a 1, the other to transmit a 0. This state difference serves as the input of the magnifier gadget. Our magnifier gadgets use this state as input to create a cascading time difference, typically (but not exclusively) by creating a large number of cache misses for one of these two states, and a large number of cache hits for the other. We first introduce two methods to compose a magnifier gadget in a (widely used [33]) tree-based pseudo-last-recently-used (PLRU) L1 cache – one for the P/A transient racing gadget, and one for the reorder-based non-transient gadget. Finally, we illustrate magnifier gadgets on both a cache with arbitrary replacement policy, and one not using the cache at all – showing that changing the replacement policy is no cure for Hacky Racers, even if specific policies may make the gadgets faster and simpler to implement.

Note that our magnifier gadgets simply transform a particular state difference to a large timing difference; this is not limited to being from the output of the racing gadgets from section 5. Indeed, the first gadget is taken from a Spectre attack [62] in the literature, which generates the state difference directly. Rather, the racing gadgets are used to convert arbitrary expressions to a particular state-change format, as input to a given magnifier. This generic timing capability can manifest in subtle ways: for example in section 7.4, where a racing gadget allows us to convert a last-level cache side channel into an L1-cache side channel, thus allowing us to use an L1-cache magnifier and thus demonstrate the first eviction-set generator in JavaScript without SharedArrayBuffer.

6.1 PLRU Gadget for Presence/Absence (P/A) Input

This first magnifier gadget is inspired by Röttger and Janc [62], who used a similar strategy as part of the leaky.page proof-of-concept of a Spectre V1 JavaScript attack. We repurpose it to time arbitrary execution. It utilizes the tree-based Pseudo-Least-Recently-Used (PLRU) [33] cache replacement policy, prevalent on modern CPUs [62], to amplify the timing difference of a single victim access. PLRU policies implement a binary tree structure to approximate a Least Recently Used (LRU) replacement policy. For a clear illustration, we take the PLRU cache’s set associativity \(W = 4\) as an example, and assume all of the following accesses are mapped into the same cache set. As is shown in figure 3, each leaf node represents a cache line within that set, and the letter within that node indicates the data currently filled in that line. Cache misses are marked in red and those arrows from the root to the leaf always composes one path pointing to the eviction candidate. Every time an access happens to a specific location within that set, it will flip arrows on its path.

The idea behind this gadget is that if the target memory address is brought into the cache by the racing gadget, it will be kept in the L1 cache for the entire magnifier gadget, such that the observed capacity of the set (in timing terms) is less by one than the elements of the following access pattern. Here we construct the access pattern to be a repetition of four locations: (B, C, E, C, D, C). If A is absent from the cache, no miss will occur. If it is present, the PLRU gadget will never evict it, and thus the cache state and the access pattern will repeatedly go back to the previous state in figure 3.8, forming a cycle from figure 3.3 to 3.8. Therefore, cache misses happen every other access, as is shown in figures 3.2, 3.4, and 3.6.

6.2 PLRU Gadget for Reorder Input

This gadget takes advantage of the fact that A will be put in different locations within the cache set based on whether A or B is accessed.
The initial cache state is shown in fig 3.1. Before A was accessed, B was the eviction candidate (EVC), and thus is replaced by A. The pseudo-LRU state is flipped down the accessed path, causing the EVC to switch to E (fig 3.2). In fig 3.3, since access C is a cache hit, nothing is evicted, however the EVC still changes, flipping the arrows down the accessed path between it and fig 3.4.

After accessing E in fig 3.4, A becomes the new EVC (fig 3.5). Since we want to keep A in the cache to allow repeated measurement, C is accessed next, making B the new EVC in fig (3.6), causing D to evict B rather than A.

(c) Since A becomes the EVC again in fig 3.7, C needs to be accessed again to flip the pseudo-LRU state at the top of the tree. Now both the cache state and the access pattern go back to that in fig 3.2, allowing the sequence to be repeated indefinitely without a new access to A.

The following accesses pattern is a repetition of (C, E, D, C, B). Figure 3 and 4 contrast the cache states of two different relative orders between A and B during the execution of the magnifier gadget. Both start with the same initial state, but proceed differently depending on how A and B are reordered. When A is accessed before B, the state-change flow is the identical to that in the P/A PLRU gadget, despite requiring no transient execution and thus meaning the racing gadget accesses exactly the same cache lines regardless of transmitted signal. The difference is subtle: B in the Reorder Input magnifier gadget is a part of the racing gadget, whereas it is part of the magnifier gadget in the P/A version. By contrast, when B is accessed first, A will be evicted after several accesses, as is shown in fig. 4. After that, there will be no more cache misses, since all following accesses fit into the cache.

6.3 Arbitrary Replacement Gadget for P/A Input

To demonstrate that changing the replacement policy will not eliminate this exploit, we introduce another magnifier gadget for caches with an arbitrary replacement policy within each set. This gadget also utilizes ILP, using a chain reaction to amplify the timing difference of probabilistic state changes across many sets, rather than only converting the timing difference into a deterministic cache state change.

The basic strategy involves the magnifier gadget itself becoming a racing gadget, with two paths that exhibit no contention when closely aligned in execution time, and heavy contention when misaligned (e.g. due to the unavailability of an input delaying one path). Once these paths become misaligned, each repetition stays...
misaligned, multiplying the delay for each stage to complete. To simplify the demonstration, in the following we focus on an L1 cache with 64 sets, 8 ways and a random replacement policy [14].

The attacker first prepares the initial cache state across N (N < 64) chosen sets, with W ways per set. The number N is linear to the final timing difference achieved, limiting the magnification achievable through sets alone, but we also develop a prefetching mechanism to further magnify this timing difference indefinitely. Two paths, PathA and PathB, are generated in the attacker’s code. We define both SEQi and PARi to each be subsets of addresses within the eviction sets of the ith cache set, without overlap between them. PARi should be sized such that bringing its elements into the cache should evict at least one member of SEQi.

Before PathA or PathB start, we fill the N chosen sets with data from SEQi, ensuring that every element of SEQi is within the cache6. The memory-access pattern in PathA and PathB is shown in figure 5a. When i is odd, SEQi is accessed by PathB. When i is even, SEQi is accessed by PathA, followed by PARi+1. Arrows in figure 5a represents the data dependence that partially restricts the actual access order during runtime. For example, memory accesses within SEQ0 are executed sequentially, while the accesses within the PARi that immediately follow can be issued after the last access in SEQ0, in parallel with each other. This access pattern ensures that the critical path of both PathA and PathB is only constructed from accesses in SEQi, as the SEQi accesses will all hit in the cache, and so both paths should finish at approximately the same time when there is no interference between them.

By contrast, consider the timeline shown in figure 5b, when PathA starts earlier than PathB. By the time PathB starts loading SEQ1, PathA has already accessed memory from PARi, which (potentially randomly) evicts elements in SEQ1. Thus, one or several cache misses will occur when accessing SEQ1. While the precise number of misses depends on the replacement policy, this will then delay the start of SEQ2, which means that PathA’s PARi will create misses in PathB’s SEQi. This delay accumulates over multiple rounds, with each round with a successful miss increasing the delay.

**Path prefetching.** As the number of cache misses induced in figure 5b is linear to the number of cache sets, the amplification rate is limited, and so timers of coarser granularity would mitigate the attack mechanism. To succeed in spite of these, we can reuse the finite number of cache sets to generate theoretically unlimited timing difference, by generating a cycle of prefetches (looping through SEQi) within the paths to reformulate the initial conditions that caused the timing difference.

Specifically, because the initial cache state is destroyed after being accessed by PARi from PathA in both the case of figure 5a and 5b, we add parallel prefetch instructions on PathB’s non-critical path with a prefetch distance DIST, as is shown in figure 5c. These may be either software prefetch instruction or standard loads – neither will block the out-of-order pipeline, and so neither will affect the execution time of PathB’s critical path, save for the intended interference7. This prefetching prepares the initial state for sets that will be later accessed by PathB.

**What if PathA runs too far ahead of PathB?** As there is no data dependence between the two paths, in figure 5b and 5c PathA could execute far ahead of PathB as time goes by since there is no miss in its critical path. Although this could cause a processor stall as instructions from PathA and PathB fill the reorder buffer, it has little effect on the final timing. This is because the slower path, PathB, continues execution during the stall, and PathA will still run ahead of PathB after the stall ends. Likewise, PathA starting too early has the same effect as both starting at the same time; due

---

6 Even with random replacement, this initial state can be achieved through repeatedly accessing SEQi, provided SEQi is smaller than the set size.

7 Note that a software prefetch, especially if marked as non-temporal, might be inserted into ways that are easier to be evicted, making the timing difference easier to accumulate [62].
to limited ROB capacity, Path_A cannot run ahead enough in this case to cause interference from different stages.

How many number of accesses should be included in each SEQ_i and PAR_i? Larger sizes of SEQ_i and PAR_i will bring higher chance of cache misses, provided SEQ_i fits in the cache. We found that for a random replacement policy, with SEQ_i set to 6 accesses (three quarters the associativity), 5 addresses in PAR_i was sufficient to provide at least 1 cache miss in SEQ_i with a 96% chance, with larger values of either increasing the chance to near certainty. A failure to achieve a cache miss, beyond the first round, does not cause the attack to fail: it only causes a single round to not add any further delay. The subsequent round will still be delayed if any previous rounds were, and so will still be able to magnify the delay further.

6.4 Arithmetic-Operation-Only Gadget for P/A Input

Although it is possible for previous magnifier gadgets to succeed on cache-based channels, we provide a further gadget that makes no use of the cache whatsoever. Since this gadget is composed of arithmetic operations, it eludes any arbitrary form of cache defence.

Similar to the gadget in section 6.3, chain reaction and ILP are also exploited in this gadget by composing two paths, Path_A and Path_B. The difference is that sequential accesses are replaced by sequential arithmetic operations, and the contention in cache capacity is replaced by contention in not-fully-pipelined functional units (e.g. dividers). This magnifier is somewhat inspired by SpectreRewind [25] and Speculative Interference [15], which use arithmetic-unit contention to generate Spectre side channels. However, here we re-purpose such contention to generate much larger timing differences from a state generated via racing gadget: thus rather than a single timing difference, we must spawn a chain reaction.

As is shown in figure 6, Path_A is constructed by chained integer multiply operations (MULs), chained integer add operations (ADDs), and parallel floating point divide operations (DIVs), while Path_B is constructed from chained DIVs and ADDs. Without contention, the latency of each arithmetic operation is fixed. For simplicity, we assume LatencyDIV = 9 cycles, LatencyMUL = 3 cycles and LatencyADD = 1 cycles. The MUL and DIV operations serve as the racing stage, while the ADDs serve as a buffering stage.

When both paths start execution at the same time in figure 6a, we would like the racing stage to finish at the same time. Since the latency of each DIV is three times that of a MUL, we set the number of MUL operations three times that of DIV. Therefore, the parallel DIVs after the racing stage in Path_A will not delay the execution time of that in Path_B. In addition, we set the number of ADDs in each ADD chain to be the same as each other, and large enough so that the next racing stage will start at the same time, after all parallel DIVs have finished executing. As most CPUs can execute at least two ADDs at the same time, there is no interference within this buffer stage.

In contrast, figure 6b shows how contention cascades when Path_B starts later. Since the execution time of their racing stages is the same, the parallel DIVs are issued before Path_B’s DIVs complete, and thus the two paths compete for resources. This results in Path_B starting the ADD buffering stage later. As the number of ADDs is the same in both paths, Path_B will then propagate this increased delay into the next racing stage against Path_A, which will grow with successive stages.

What will happen if Path_A runs too far ahead of Path_B? It is possible that Path_A’s parallel DIVs finish execution even before Path_B’s racing stage starts. In this case, Path_B will no longer suffer from the DIV unit’s contention, and the timing difference fails to increase. To solve this problem, we increase the number of operations in the racing stage, so that the limited capacity of the Reorder buffer (ROB) will stop this from occurring. Specifically, when Path_B has not started, no MUL operation in the same racing stage can release its ROB entry. As long as the number of MUL operations exceeds the ROB capacity, the processor will stall before any parallel DIV can be issued, and can only continue executing in Path_A after Path_B also starts its racing stage. This stall only delays Path_A, so the timing difference will not be affected, as Path_B is the critical path whose timing is observed.

7 ATTACKS

Here we demonstrate the utility of Hacky Racers, by showing that simple repetition attacks alone do not typically give a timing difference that can be measured, but racing gadgets can fix them. Higher quality, faster bit-rate timers can be created by combining racing gadgets and magnifier gadgets, to allow extremely precise timing (to the nanosecond granularity) to be achieved even with the attack only being repeated once. Finally we use Hacky Racers to construct a novel backwards-in-time Spectre attack [37], demonstrate its efficacy even in JavaScript, and show how Hacky Racers can be used as a drop-in replacement for SharedArrayBuffer timers [69] by using them to construct LLC eviction sets.
**Processor details** We evaluate on an Intel i7-8750H Coffee Lake processor, though we reproduced similar results when we tried them on an AMD Ryzen 5900X. The Intel system has 6 physical cores running at 2GHz. Each core has a private 32KB L1 Instruction Cache, 32KB L1 Data Cache, a 256KB L2 cache and a 64-entry L1 TLB. All cores share a 9MB L3 cache and a 1536-entry L2 TLB. The DIVSD instruction has a latency of 13-14 cycles based on the operand content, and 4-cycles reciprocal throughput [24].

Nothing we implement is particularly microarchitecture-specific: our experimental artefact works across all Intel systems we tested out-of-the-box, and only the Prime-And-Scope attack (section 7.4) fails on AMD systems, due to not having support for non-inclusive last-level caches within the underlying side-channel attack (rather than the Hacky Racer). Fundamentally the core needs to be out-of-order (i.e. any contemporary processor except the LITTLE part of Arm’s big.LITTLE cores), and some of our individual attacks (but not all Hacky Racers) require either a set-associative cache or some non-fually-pipelined execution units. All of these are ubiquitous. In general we expect some attack-core combinations will require a simple profiling stage within the attacker’s JavaScript.

### 7.1 Repetition Gadgets with Racing Gadgets

Here we first give an example of how timing difference fails to accumulate by simple repetition [45] of the flush+reload [93] process. This is an arbitrary choice of attack, chosen to demonstrate that, counter-intuitively, a simple repetition can fail in some circumstances due to attack setup time, unless magnifiers are used to generate coarse timing difference instead.

The load stage will either access the same, or a different, address as the reload stage, which will be evicted in the flush stage. As is shown in figure 7a, although the timing difference accumulates on the reload stage as expected, the loading stage has an opposite timing difference, eliminating the overall timing difference.

This makes such a simple repetition ineffective as an arbitrary timing gadget; however, Hacky Racers present a solution. We put the load stage into one path of a racing gadget, while the other path’s execution time stays constant and always costs more time than the load stage. In this case, the timing difference from the load stage is hidden and the timing side-channel reappears in the total run-time, as is shown in figure 7b.

McIlroy et al. [45] were able to use a repetition gadget to generate Spectre attacks [9], but the leakage rate is 10B/s, 50× lower than the magnifier-based Spectre attack in section 7.3. This is because most execution time will be timing invariant, and to accumulate a large enough difference from the change in the reload stage alone takes significant time, impacting the bit rate. By combining with a magnifier gadget, we can make almost the entire execution time variant on the secret, increasing bit rates as a result.

This means that the combination of racing and magnifier gadgets are more capable than repetition gadgets alone – as well as providing higher bit rates by making more of the execution time timing-variant: repetition alone is not always sufficient to generate a coarse time signal from an initial fine-grained timing signal. Still, racing gadgets alone are sufficient to improve a repetition gadget’s signal-to-noise ratio, if not achieve high bit rates, by hiding the timing-variant setup time within a timing-invariant racing gadget.

### 7.2 Racing-Gadget Granularity

To test the granularity of racing gadgets, we choose the integer add operation to construct the reference path in a Transient P/A Racing Gadget, as each add operation only costs 1 cycle. We pick operations with various latency to construct the target path, and check whether our reference path can distinguish paths with different number of chained operations. The slope of each line is close to ratio between the latency of target operation and reference operation. The granularity is the maximum consecutive points whose Y value stays unchanged, indicating those two target paths cannot be distinguished by their corresponding operation number in the reference path.

---

*The load stage of a Spectre v1 gadget happens inside a misspeculated branch, whose return to correct execution is dictated by the branch-calculation time, which must be longer than the load access. This makes such a construction a racing gadget natively.*
For target paths composed of 1-cycle-latency chained operations as well, such as ADD and LEAL instruction shown in Figure 8, the granularity is 1 – 3 operations. For MUL instructions (3-cycle latency), the granularity is 1 – 2 operations. Therefore, the overall minimal granularity of racing gadgets is 1 – 6 cycles (0.5 – 3ns). In this particular (Transient P/A) Racing Gadget, all instructions in the ref path are older than those in the target path (as the ref path is a misspeculated branch condition), thus the ROB capacity limits the length of the ref path to 54, which in turn limits the largest execution time that we can time to 54 cycles.

Using MUL operation, whose latency is longer than ADDs, we increase our timer’s maximum timing threshold to around 140 ADD operations, as is shown in Figure 9. However, it sacrifices granularity to around 2 – 4 ADD operations. Still, it can distinguishes the number of DIV operations perfectly as DIV’s latency is around 4 times of MUL’s.

7.3 Attack: SpectreBack in JavaScript

Rollback-based [64] and other Spectre mitigation tools that only clean up the effects of misprediction once it has happened [11, 26, 65] (systems that do not implement strictness order [10]) were broken by attacks [15, 25] that transmit the timing effect via contention. Here we propose a different method of breaking such systems, by transmitting timing information backwards in time to a Non-Transient Reorder Racing Gadget (section 5.2) via cache state. It is presented here not only as a user of the reorder racing/magnifier gadgets, but also because the attack itself is generated via a custom racing gadget, to transmit it to a reorder racing gadget that exists before any transient execution is squashed. We use the PLRU Magnifier Gadget for Reorder Input to extract the secret.

The example code for this gadget is shown in code listing 3. The attacker controls which instruction sequence (line 4 and line 6) completes first based on the value of secret data speculatively accessed at line 8. This timing difference can be converted into the relative access order of array[A] and array[B], similar to a non-transient reorder racing gadget. This order will then be served as the input for the PLRU Gadget for Reorder Input (section 6.2) at line 10. Figure 10 shows the output timing difference of this Gadget when its access pattern is repeated 4000 times.

Similar to leaky.page [62], we use this gadget to leak out-of-bounds array data in Chrome 88, and achieve a 4.3 kilobits/second leakage rate with over 88% accuracy (more than ample to leak secret keys).
Timing difference (s) with MUL operations can provide a fine enough granularity. We demonstrate that Hacky Racers can be conveniently utilised by various attacks, we use a combination of the (P/A) Racing Gadget and thus resurrects existing attacks even in hardened browsers.

This is just one example that shows Hacky Racers can easily be retained their 100% success rate after the timer is replaced by ours.

L3 cache miss algorithm only needs to distinguish between an L1 cache hit and an L3 cache miss, composing the reference path in the Racing Gadget. Purnal et al. [52], replacing just the timer. Since the timer in this section 2.2). We use the EV profiling algorithm of EV, which allow the setup of various other attacks (section 6.3 and 6.4) can also defeat coarsened timers, currently 5µs, in browsers.

For the Arbitrary Replacement Gadget, we set half of the L1D cache’s set number, 32, as the set number for each round. We set the prefetch distance as 22 iterations ahead. Figure 11 shows that timing difference accumulates to 100 µs as we repeatedly traverse through the sets; we also tried the arbitrary replacement gadget without prefetching, but as this was limited by the number of sets in the cache, it could only magnify up to 450 cycles (approximately 225 ns).

For the Arithmetic-operation Only Gadget, as is shown in Figure 12, the timing different stops increasing at around a repeat number of 15000. This is because the total run-time approaches the interval of timer interrupts (4 ms), and since this magnifier is entirely stateless, it cannot keep accumulating timing difference once the pipeline is reset.

Both of these are more than ample to defeat current timer coarsening. Still, unlike the PLRU-based gadgets, whose magnification rates are almost arbitrary, these magnifiers are limited to rates close to the maximum ever granularity implemented in a browser (100 ms [60]). In practice this is little impediment; both can be combined with repetition gadgets (increasing bit rate compared to repetition alone, by making more of the execution timing-variant, and avoiding any negative correlation with setup time masking the time difference), at the expense of needing the attack to be performed multiple times. Whether either or both can have their magnification rates improved, allowing the attack to be repeated only once even for arbitrarily coarse timers, is open future work.

8 POTENTIAL COUNTERMEASURES

Transient execution is not required for racing or magnifier gadgets: only the transient presence-absence (P/A) racing gadgets in section 5.1 (because they use transient execution directly) and the SpectreBack attack in section 7.3 (because it is a Spectre attack, though it does break Spectre defences [11, 26, 64, 65] that do not guarantee Strictness Order [10], and thus try to clean up misspeculation after it has occurred), use transient execution in any way, and thus only these can be guarded by Spectre defences [10, 11, 26, 34, 64–66, 84, 89, 94, 95], which are designed to eliminate the negative effects of transient execution. We use transient (P/A) gadgets in many of our examples because they are simple to understand: however, an attacker can easily change to use reorder gadgets instead.

Take DoM [66] (delay-on-miss), a technique which delays cache misses in the L1 until they become non-speculative, and the non-transient reorder racing gadget as an example. The reorder gadget only relies on the relative cache insertion order of two load instructions. Both of these can be entirely non-speculative, and will still race due to instruction-level parallelism, and yet DoM marks them as being safe to execute in any order. To take another example, GhostMinion [10] might change the original L1 cache insertion order by by first inserting speculative load into the GhostMinion cache, the exact state that ends up being inserted into the L1 cache still ends up dictated by the out-of-order execution present in the
reorder gadget\textsuperscript{9}. The fundamental property is that Spectre defences treat transient execution as the dangerous part; and indeed while it is more dangerous than Hacky Racers alone (which leak timing information rather than secrets directly), they do not seek to hide or eliminate channels caused via instruction-level parallelism.

In general, Hacky Racers can be defeated by either preventing the racing gadget’s success in creating the state change, or the magnifier gadget’s success in replicating it – though the attacker may be able to rely on repetition rather than magnification in circumstances where this still provides a high enough signal-to-noise ratio (section 7.1). Some of our magnifiers are limited in magnification capability, and could be defeated via further coarsening (section 7.5), whereas others (the PLRU gadgets) are unlikely to be limited without removing any source of coarse-grained time completely, which is likely to be impossible in practice. We expect coarser-grained magnifiers not using PLRU to follow, and even our least effective magnifiers would still combine well with repetition in such an environment.

One potential way to completely mitigate the cache-based reorder gadget we give in this paper is to guarantee cache state is always equivalent to an in-order execution. We leave the design, evaluation and overheads (and whether such overheads are feasible for deployment, given the existing overheads of Spectre defences without such guarantees) of such a scheme to future work. Likewise, our reorder gadget is based on PLRU replacement, and so other policies will break this specific gadget, though we present others that translate over simply. But even if this cache-based gadget is mitigated, an attacker can then change strategy to transmit timing based on within-core contention – where in the general case, assurance behavior equivalent to in-order execution is likely to require actual in-order execution.

For run-time detection mechanisms\textsuperscript{12, 17, 20, 50, 96}, we expect racing gadgets to look so similar to normal out-of-order execution that they will be difficult to catch without very high false positive rates. Still, since magnifier gadgets rely on highly repetitive patterns, detection of at least high-bitrate channels may be feasible, but will require different schemes for every possible repeated pattern, and so attackers will keep changing strategy. Frequent L1 cache misses exists in PLRU gadget of section 6.1 and Arbitrary Replacement Gadget in section 6.3, therefore the L1 cache miss counter could be utilized to as one input to such a detector, though only as a very weak classifier. Similar to the port contention attack by Rokicki et al.\textsuperscript{61}, the Arithmetic-Operation-Only Gadget of section 6.4 executes long back-end-bounded instruction (arithmetic operations) chain without misprediction, which makes the ratio of back-end-bound execution divided by misprediction-bound execution also a potential parameter to detect this gadget. Besides, since this attack also requires precise instruction sequence construction like that by Rokicki et al.\textsuperscript{61}, the reordering of instruction sequences from either browser’s optimization or software-diversification\textsuperscript{23, 55}

can affect the efficiency of Hacky Racers, and software analysis\textsuperscript{85} within JavaScript compilers may be able to pick up attacks that are less well obfuscated. Still, in some scenarios attackers may have enough control to manually construct the instruction sequence or implement multiple backup sequences to overcome any such analysis. Again, we leave this for future study.

For browser security, where the main threats are from making Spectre and fingerprinting attacks simpler even without SharedArrayBuffer timers, cross-origin isolation policies\textsuperscript{36} (as opposed to site isolation alone which was attacked by spook.js\textsuperscript{7}) move many targets outside the address space, although implementation requires manual effort to avoid breaking compatibility, and so adoption is not yet universal. A similar process-level isolation policy\textsuperscript{4} is needed in Cloudflare, otherwise Hacky Racers will make the Spectre-attack proposed by Martin et al.\textsuperscript{71} much easier. We also believe that Tor, which currently disables SharedArrayBuffer even with cross-origin isolation\textsuperscript{3}, now gains very limited protection from this constraint, in the presence of Hacky Racers.

9 DISCUSSION AND CONCLUSION

Hacky Racers show that high-resolution timing information can be extracted via instruction-level parallelism, even in highly restricted JavaScript environments, and even without multiple threads\textsuperscript{69} or any language features that could realistically be disabled. We do not believe mitigation to be particularly realistic; out-of-order execution is vital for single-threaded performance (so systems will not be going in-order), removal of PLRU cache replacement\textsuperscript{33} will only cause the attacker to change strategy, and while some of our gadgets could be eliminated through further coarsening, others work to almost arbitrary degree, and all can be amplified further by repetition. We also expect further magnifiers to follow. This demonstrates that timer coarsening has limited efficacy in any browser security model, and the removal of SharedArrayBuffer from sites without cross-site isolation\textsuperscript{13} in Chrome and Firefox is insufficient to protect them from timing side channels. Threats will have to be dealt with, in future, by isolation\textsuperscript{57} only. Hacky Racers may also impact other security models with restricted timing, such as the removal of user-privilege fine-grained timers on M1 processors and other ARM systems\textsuperscript{41, 56}; we leave consideration of these to future work.

Our attacks also have implications for architects. While Spectre\textsuperscript{37} showed speculative execution to be a fundamental source of information leakage, Hacky Racers goes even further: even correct execution results in information leakage. Is any microarchitectural performance optimisation truly secure, given the right threat model?

DATA AVAILABILITY STATEMENT

Source code for our attacks and gadgets is available in our data repository\textsuperscript{87}, and is described in detail in the below appendix.

ACKNOWLEDGEMENTS

We would like to thank our ASPLOS 2023 Reviewers for their helpful feedback. We would also like to thank Thomas Rokicki for providing a modified version of Chromium.
A ARTIFACT APPENDIX

A.1 Abstract
Our artifact provides source code and scripts for four timing side-channel experiments mentioned in this paper, corresponding to sections 7.3 to 7.5, and each demonstrates the efficiency and portability of either/both Racing and Magnifier Gadgets. Our results should be evaluated on an Intel or AMD x86 machine (we used an i7-8750h, but systems of a similar architecture will work out-of-the-box). Migration to systems with other ISAs or significantly different microarchitectures require minor source-code modifications.

A.2 Artifact Check-list (Meta-information)

- **Algorithm**: ILP-based magnifiers, SpectreBack attack & Hacky Racer-integrated Prime+Scope [52].
- **Program**: Arbitrary Replacement Magnifier Gadget (sec 6.3), Arithmetic-Operation-Only Magnifier Gadget (sec 6.3), SpectreBack Attack & Prime+Scope-based LLC Eviction Set profiling.
- **Compilation**: Succeeded on all tested compilers (gcc-6,7,8,10 & clang 10).
- **Binary**: Can be generated from script and source code.
- **Run-time environment**: Linux shell scripts.
- **Hardware**: ILP-based magnifiers succeeded on tested Intel (Skylake, Haswell, Coffee Lake) and AMD (Zen 2, Zen3) CPUs. SpectreBack attack & Hacky Racer-integrated EV set profiling are successfully tested on an i7-8750h CPU. Other CPUs should give similar results.
- **Run-time environment**: Provided scripts are for Linux, but the source code and script could be applied to others with minor modifications. The SpectreBack attack requires Chrome 88 (can be altered to other browsers by changing parameters), Python, and flask packages.
- **Output**: Accumulated timing difference versus magnify rounds for those two magnifier gadgets. The data points should be similar to that in figure 11 and 12. A screenshot of the expected output of SpectreBack attack is stored in the Github repo. The output of Hacky Racer-integrated Prime+Scope should have the same output to the original Prime+Scope, showing whether an LLC EV is successfully generated or not.
- **Publicly available?**: Yes, at [https://zenodo.org/badge/latestdoi/551660156](https://zenodo.org/badge/latestdoi/551660156).

A.3 Description

A.3.1 How to access. Our source code and scripts are available on Github: [https://github.com/FxPiGaAo/Hacky-Racer](https://github.com/FxPiGaAo/Hacky-Racer)

A.3.2 Hardware dependencies. We recommend to run our testing on an Intel Coffee Lake (an i7-8750H was our main test system, but other recent Intel systems should deliver similar results) or AMD ZEN 2/3 (we used Ryzen 5900hx). For the Hacky Racer-integrated EV-set profiling attack, our prototype requires an inclusive LLC (as featured on most recent Intel cores, but not on Ryzen cores), so AMD systems can only be used to replicate the other three attacks out-of-the-box.

A.3.3 Software dependencies. Our source code assumes Ubuntu 18.04 on x86-64 systems. Other Linux-based operating systems should also work. To run the SpectreBack attack experiment, python, flask and Chrome 88 are required to be installed (instructions given in the repository).

A.4 Installation
You can clone our source code and scripts from Github: `git clone https://github.com/FxPiGaAo/Hacky-Racer.git`

A.5 Experiment Workflow
Before running any experiments, we recommend to fix the CPU frequency following the instruction given in the README.md to eliminate the affect from DVFS. Any value is suitable (and the attacks will succeed without doing this, but with more noise), but we ran at 2GHz. For both the Arbitrary Replacement Magnifier Gadget and SpectreBack attack, multiple attempts may be required for success, as background processes might interfere with the cache and affect the result.

To run the Arithmetic-Operation-Only Magnifier Gadget, enter the `Arithmetic-Operation-Only Magnifier Gadget directory and execute ./test.sh`. This test should take around 1min.

To run the Arbitrary Replacement Magnifier Gadget, enter the `Arbitrary Replacement Magnifier Gadget directory and execute make: /arbitrary`. This test should take around 30s.

To run the SpectreBack attack, please follow the instruction provided in `spectre.js/README.md step-by-step to install the prerequisites (chrome 88 & flask) and execute the attack. This test should take around 20 minutes to set up the requirements and another 5 minutes to go through the attack.

To run the Prime+Scope integrated with Hacky Racers, please follow the instructions given in `PRIME-SCOPE-hacky racer/README.md`.

A.6 Evaluation and Expected Results
The expected result of each experiments are put into the README.md of each experiment’s folder.

A.7 Experiment Customization
For the SpectreBack attack, minor code modifications can be applied to run it on different versions of Chrome or other browsers (as with the [https://leaky.page/](https://leaky.page/) code is it based on). Specifically, only offset parameters at lines 103 - 105 are specific to Chrome 88.

For the Arbitrary Replacement Magnifier Gadget, parameters such as cache offset, prefetch distance, and number of cache accesses within each set per round, can be easily modified in the source code.

For the Arithmetic-Operation-Only Magnifier Gadget, path lengths can be modified in either the `magnify()` in arithmetic_source.c or the `test.sh`. The operation type in the racing stage can also be changed by modifying the assembly code part in `magnify()`.

For the Prime+Scope integrated with Hacky Racers, a magnifier gadget can be implemented after the racing gadget through source code modification if necessary.

A.8 Methodology
The artifact for this paper was reviewed according to the guidelines at [https://sites.google.com/stanford.edu/asplos23a0/home](https://sites.google.com/stanford.edu/asplos23a0/home).
REFERENCES

[1] 2015. Improve Time Resolution Defense. https://gitlab.torproject.org/tpo/applications/tor-browser/-/issues/16110.

[2] 2018. Mitigating Side-Channel Attacks. https://www.chromium.org/home/security/privacy/security.

[3] 2020. Enable SharedArrayBuffers with sequential threads? https://torproject.org/torbug/issue/40016.

[4] 2022. Cloudflare Workers. https://developers.cloudflare.com/workers/learning/how-workers/workers/.

[5] 2022. High Resolution Time. https://www.w3.org/TR/hr-time/.

[6] 2022. Timer in Chrome. https://chromestatus.com/feature/6497206758599264.

[7] Ayush Agarwal, Soili O’Connell, Jason King, Shaed Yeheskeli, Daniel Genkin, Eyal Ronen, and Yuval Yarom. 2022. Spook.js: Attacking Chrome Strict Site Isolation via Speculative Execution. In 2022 IEEE Symposium on Security and Privacy (SP).

[8] Pavlos Aimoniotis, Christos Sakalis, Magnus Sjlander, and Stefanos Kaxiras. 2021. "It’s a Trap!"–How Speculation Invariance Can Be Abused with Forward Speculative Interference. arXiv preprint arXiv:2109.10774 (2021).

[9] Pavlos Aimoniotis, Christos Sakalis, Magnus Sjlander, and Stefanos Kaxiras. 2021. Reorder Buffer Contention: A Forward Speculative Interference Attack for Speculation Invariant Instructions. IEEE Computer Architecture Letters 20, 2 (2021).

[10] Sam Ainsworth. 2021. GhostMinion: A Strictness-Ordered Cache System for Spectre Mitigation. In MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture. 592–606.

[11] Sam Ainsworth and Timothy M Jones. 2020. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 132–144.

[12] Manaar Alam, Sarani Bhattacharyya, Deebop Mohudhop, and Sourangshu Bhattacharya. 2017. Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel attacks. Cryptology ePrint Archive (2017).

[13] Jake Archibald and Eiji Kitamura. 2022. https://developer.chrome.com/blog/enabling-shared-array-buffer/.

[14] ARM Limited. [n.d.]. ARM1176JZF-S Technical Reference Manual. https://developer.arm.com/documentation/dd101b/level-one-memory-system/cache-organization/features-of-the-cache-system/lang-en.

[15] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jinyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Yarom, dumpster, et al. 2021. Spectre interference attacks: Breaking invisible speculation schemes. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 1046–1060.

[16] Ari Bhattacharyya, Alexandra Sandulescu, Matthias Neugebauer, Alessandro Soriozzi, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. sMoTher: Spectre: exploiting speculative execution through port contention. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 769–785.

[17] Samira Brongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. 2018. Hacky Racers: Exploiting Instruction-Level Parallelism to Generate Stealthy Fine-Grained Timers. ASPLOS ’23, March 25–29, 2023, Vancouver, BC, Canada.

[18] Samira Brongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. 2021. Replicating and Mitigating Spectre Attacks on an Open Source RISC-V Microarchitecture. In CARRV.

[19] Ben Gras, Kaveh Razavi, Erik Bosman, Herman Pos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In NDSS. Vol. 17. 26.

[20] Daniel Gruss, Clementine Maurice, Andreas Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. 368–379.

[21] Daniel Gruss, Clementine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A remote software-induced fault attack in javascript. In International conference on detection of intrusions and malware, and vulnerability assessment. Springer, 309–321.

[22] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache template attacks: Automating attacks on inclusive last-level caches. In 24th USENIX Security Symposium (USENIX Security 15) 897–912.

[23] ECMA International. 2017. ECMA-Script 2017 Language Speciﬁcation. https://www.ecma-international.org/wp-content/uploads/ECMA-262_8th_edition_june_2017.pdf.

[24] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S S A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In 2015 IEEE Symposium on Security and Privacy. IEEE, 591–604.

[25] Kamil Kędzierski, Miquel Moreno, Francisco J Caizarla, and Mateo Valero. 2010. Adapting cache partitioning algorithms to pseudo-ru replacement policies. In 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). IEEE, 1–12.

[26] Khaled N Khawaseen, Emaeel Mohammadian Koruye, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. SafeSpec: Banning the spectre of a meltdown with leakage-free speculation. In 2019 56th ACM/IEEE Design Automation Conference (DAC). IEEE, 1–6.

[27] Vladimir Kiriansky and Carl Walldt. 2018. Speculative buffer overflows: Attacks and defenses. arXiv preprint arXiv:1807.03577 (2018).

[28] Eiji Kitamura and Domenic Demicola. 2021. Why you need "cross-origin isolated" for powerful features. https://web.dev/why-coop-eco/.

[29] Paul Kocher, Jann Horn, Andreas Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 1–19.

[30] David Kohlbrenner and Hovav Shacham. 2016. Trusted browsers for uncertain times. In 25th USENIX Security Symposium (USENIX Security 16). 463–480.

[31] Emaeel Mohammadian Koruye, Khaled N Khawaseen, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre results! speculation attacks using the return stack buffer. In 12th USENIX Workshop on Offensive Technologies (WOOT 18).

[32] Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. AMD Prefetch Attacks through Power and Time. In 31st USENIX Security Symposium (USENIX Security 22).

[33] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clementine Maurice, and Stefan Mangard. 2016. (ARMageddon): Cache Attacks on Mobile Devices. In 25th USENIX Security Symposium (USENIX Security 16). 549–564.

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

[35] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In 2015 IEEE symposium on security and privacy. IEEE, 605–622.

[36] Giorgio Masiuradze and Christian Rösaw. 2018. re2spec: Speculative execution using return stack buffers. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2109–2122.

[37] Ross McLizzy, Jaroslav Sevcik, Tobias Tebhi, Ben L Titser, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. arXiv preprint arXiv:1902.05178 (2019).

[38] MDN Contributors. 2022. High resolution time. https://developer.mozilla.org/en-US/docs/Web/API/Performance_API.

[39] MDN Contributors. 2022. performance.now(). https://developer.mozilla.org/en-US/docs/Web/API/Performance_now.

[40] 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 Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. 1406–1418.

[41] Dag Arne Ostvik, Aðal Shumi, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In Cryptographers’ track at the RSA conference. Springer, 1–20.

[42] Mathias Payer. 2016. HexPADS: a platform to detect “stealth” attacks. In International Symposium on Engineering Secure Software and Systems. Springer, 138–154.

[43] Colin Percival. 2005. Cache missing for fun and profit. In International Conference on Computer Security and Privacy. Springer.

[44] Antono Purnal, Furkan Turan, and Ingrid Verbauwhede. 2021. Prime+ Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications.
