Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript

Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard

Graz University of Technology, Austria

Abstract. Research showed that microarchitectural attacks like cache attacks can be performed through websites using JavaScript. These timing attacks allow an adversary to spy on users secrets such as their keystrokes, leveraging fine-grained timers. However, the W3C and browser vendors responded to this significant threat by eliminating fine-grained timers from JavaScript. This renders previous high-resolution microarchitectural attacks non-applicable.
We demonstrate the inefficacy of this mitigation by finding and evaluating a wide range of new sources of timing information. We develop measurement methods that exceed the resolution of official timing sources by 3 to 4 orders of magnitude on all major browsers, and even more on Tor browser. Our timing measurements do not only re-enable previous attacks to their full extent but also allow implementing new attacks. We demonstrate a new DRAM-based covert channel between a website and an unprivileged app in a virtual machine without network hardware. Our results emphasize that quick-fix mitigations can establish a dangerous false sense of security.

1 Introduction

Microarchitectural attacks comprise side-channel attacks and covert channels, entirely implemented in software. Side-channel attacks exploit timing differences to derive secret values used in computations. They have been studied extensively in the past 20 years with a focus on cryptographic algorithms [2,10,16,29–31,48]. Covert channels are special side channels where a sender and a receiver use the side channel actively to transmit data covertly. These attacks require highly accurate timing and thus are typically implemented in native binaries written in C or assembly language to use the best available timing source.

Side channels exist on virtually all systems and software not hardened against side channels. Thus, browsers are an especially easy target for an attacker, because browsers process highly sensitive data and attackers can easily trick a victim to open a malicious website in the browser. Consequently, timing side-channel attacks have been demonstrated and observed in the wild, to recover a user's browser history [8, 13, 41], but also a user's geolocation [14], whether a user is logged in to another website [4] and even CSRF tokens [11]. Van

Goethem et al. [37] exploited more accurate in-browser timing to obtain information even from within other websites, such as contact lists or previous inputs.

Oren et al. [28] recently demonstrated that cache side-channel attacks can also be performed in browsers. Their attack uses the `performance.now` method to obtain a timestamp whose resolution is in the range of nanoseconds. It allows spying on user activities but also building a covert channel with a process running on the system. Gruss et al. [9] and Bosman et al. [5] demonstrated Rowhammer attacks in JavaScript, leveraging the same timing interface. In response, the W3C [40] and browser vendors [1, 3, 6] have changed the `performance.now` method to a resolution of 5 µs. The timestamps in the Tor browser are even more coarse-grained, at 100 ms [25]. In both cases, this successfully stops side-channel attacks by withholding necessary information from an adversary.

In this paper, we demonstrate that reducing the resolution of timing information or even removing these interfaces is completely insufficient as an attack mitigation. We propose several new mechanisms to obtain absolute and relative timestamps. We evaluated 10 different mechanisms on the most recent versions of 4 different browsers: Chrome, Firefox, Edge, as well as the Tor browser, which took even more drastic measures. We show that all browsers leak highly accurate timing information that exceeds the resolution of official timing sources by 3 to 4 orders of magnitude on all browsers, and by 8 on the Tor browser. In all cases, the resolution is sufficient to revive the attacks that were thought mitigated [28].

Based on our novel timing mechanisms, we are the first to exploit DRAM-based timing leaks from JavaScript. There were doubts whether DRAM-based timing leaks can be exploited from JavaScript, as it is not possible to directly reach DRAM [32]. We demonstrate that a DRAM-based covert channel can be used to exfiltrate data from highly restricted, isolated execution environments that are not connected to the network. More specifically, we transmit data from an unprivileged process in a Virtual Machine (VM) without any network hardware to a website, by tunneling the data through the DRAM-based covert channel to the JavaScript running in a web browser on the same host machine.

Our key contributions are:

- We performed a comprehensive evaluation of known and new mechanisms to obtain timestamps. We compared 10 methods on the 3 major browsers on Windows, Linux and Mac OS X, as well as on Tor browser.
- Our new timing methods increase the resolution of official methods by 3 to 4 orders of magnitude on all browsers, and by 8 orders of magnitude on Tor browser. Our evaluation therefore shows that reducing the resolution of timer interfaces does not mitigate any attack.
- We demonstrate the first DRAM-based side channel in JavaScript to exfiltrate data from a highly restricted execution environment inside a VM with no network interfaces.
- Our results underline that quick-fix mitigations are dangerous, as they can establish a false sense of security.

The remainder of this paper is organized as follows. In Section 2, we provide background information. In Section 3, we comprehensively evaluate new timing measurement methods on all major browsers. In Section 4, we demonstrate the revival of cache attacks with our new timing primitives as well as a new DRAM-based covert channel between JavaScript in a website and a process that is strictly isolated inside a VM with no network hardware. Finally, we discuss effective mitigation techniques in Section 5 and conclude in Section 6.

2 Background

2.1 Microarchitectural attacks

A large body of recent work has focused on cross-VM covert channels. A first class of work uses the CPU cache for covert communications. Ristenpart et al. [33] are the first to demonstrate a cache-based covert channel between two Amazon EC2 instances, yielding 0.2 bps. Xu et al. [47] optimized this covert channel and assessed the difference in performance between theoretical and practical results. They obtain 215.11 bps with an error rate of 5.12%. Maurice et al. [23] built a cross-VM covert channel, using the last-level cache and a Prime+Probe approach, that achieves a bit rate of 751 bps with an error rate of 5.7%. Liu et al. [21] demonstrated a high-speed cache-based covert channel between two VMs that achieves transmission speeds of up to 600 Kbps and an error rate of less than 1%. In addition to the cache, covert channels have also been demonstrated using memory. Xiao et al. [46] demonstrated a memory-based covert channel using page deduplication. Wu et al. [45] built a covert channel of 746 bps with error correction, using the memory bus. Pessl et al. [32] reverse engineered the DRAM addressing functions that map physical addresses to their physical location inside the DRAM. The mapping allowed them to build a covert channel that relies solely on the DRAM as shared resource. Their cross-core cross-VM covert channel achieves a bandwidth of 309 Kbps. Maurice et al. [24] demonstrated an error-free covert channel between two Amazon EC2 instances of more than 360 Kbps, which allows building an SSH connection through the cache.

2.2 JavaScript and timing measurements

JavaScript is a scripting language supported by all modern browsers, which implement just-in-time compilation for performance. Contrary to low-level languages like C, JavaScript is strictly sandboxed and hides the notion of addresses and pointers. The concurrency model of JavaScript is based on a single-threaded *event loop* [26], which consists of a message queue and a call stack. Events are handled in the message queue, moved to the call stack when the stack is empty and processed to completion. As a drawback, if a message takes too long to process, it blocks other messages to be processed, and the browser becomes unresponsive. Browsers received the support for multithreading with the introduction of *web workers*. Each web worker runs in parallel and has its own event loop [26].

For timing measurement, the timestamp counter of Intel CPUs provides the number of CPU cycles since startup and thus a high-resolution timestamp. In native code, the timestamp counter is accessible through the unprivileged `rdtsc` instruction. In JavaScript, we cannot execute arbitrary instructions such as the `rdtsc` instruction. One of the timing primitives provided by JavaScript is the High Resolution Time API [40]. This API provides the `performance.now` method that gives a sub-millisecond timestamp. The W3C standard recommends that the timestamp should be monotonically increasing and accurate to 5 µs. The resolution may be lower if the hardware has no support for such a high resolution.

Remarkably, until Firefox 36 the High Resolution Time API returned timestamps accurate to one nanosecond. This is comparable to the native `rdtsc` instruction which has a resolution of 0.5 ns on a 2 GHz CPU. As a response to the results of Oren et al. [28], the timer resolution was decreased for security reasons [3]. In recent versions of Chrome and WebKit, the timing resolution was also decreased to the suggested 5 µs [1, 6]. The Tor project even reduced the resolution to 100 ms [25]. The decreased resolution of the high-resolution timer is supposed to prevent time-based side-channel attacks. In a concurrent work, Kohlbrenner et al. [18] showed that it is possible to recover a high resolution by observing clock edges, as well as to create new implicit clocks using browser features. Additionally, they implemented fuzzy time that aims to degrade the native clock as well as all implicit clocks.

2.3 Timing attacks in JavaScript

Van Goethem et al. [37] showed different timing attacks in browsers based on the processing time of resources. They aimed to extract private data from users by estimating the size of cross-origin resources. Stone [35] showed that the optimization in SVG filters introduced timing side channels. He showed that this side channel can be used to extract pixel information from iframes.

Microarchitectural side channels have only recently been exploited in JavaScript. Oren et al. [28] showed that it is possible to mount cache attacks in JavaScript. They demonstrated how to generate an eviction set for the last-level cache that can be used to mount a Prime+Probe attack. Based on this attack, they built a covert channel using the last-level cache that is able to transmit data between two browser instances. Furthermore, they showed that the timer resolution is high enough to create a spy application that tracks the user's mouse movements and network activity. As described in Section 2.2, this attack caused all major browsers to decrease the resolution of the `performance.now` method.

Gruss et al. [9] demonstrated hardware faults triggered from JavaScript, exploiting the so-called Rowhammer bug. The Rowhammer bug occurs when repeatedly accessing the same DRAM cells with a high frequency [15]. This "hammering" leads to bit flips in neighboring DRAM rows. As memory accesses are usually cached, they also implemented cache eviction in JavaScript.

All these attacks require a different timestamp resolution. The attacks from Goethem et al. [37] and Stone [35] require a timestamp resolution that is on the order of a microsecond, while the attack of Oren et al. [28] relies on the

fine-grained timestamps on the order of nanoseconds. More generally, as microarchitectural side channel attacks aim at exploiting timing differences of a few CPU cycles, they depend on the availability of fine-grained timestamps. We note that decreasing the resolution therefore only mitigates microarchitectural attacks on the major browsers that have a resolution of 5 µs, but mitigates more side-channel attacks on the Tor browser which has a resolution of 100 ms.

3 Timing Measurements in the JavaScript Sandbox

This section describes techniques to get accurate measurements with a high-resolution timestamp in the browser. In the first part, we describe methods to recover a high resolution for the provided High Resolution Time API. The second part describes different techniques that allow deriving highly accurate timestamps, with *implicit* timers. These methods are summarized in Table 1.

3.1 Recovering a high resolution

In both Chrome and Webkit, the timer resolution is decreased by rounding the timestamp down to the nearest multiple of 5 µs. As our measurements fall below this resolution, they are all rounded down to 0. We refer to the underlying clock's resolution as *internal resolution* and to the decreased resolution of the provided timer as *provided resolution*. It has already been observed that it is possible to recover a high resolution by observing the clock edges [18, 22, 34, 38]. The clock edge aligns the timestamp perfectly to its resolution, *i.e.*, we know that the timestamp is an exact multiple of its provided resolution at this time.

Clock interpolation As the underlying clock source has a high resolution, the difference between two clock edges varies only as much as the underlying clock. This property gives us a very accurate time base to build upon. As the time between two edges is always constant, we interpolate the time between them. This method has also been used in JavaScript in a concurrent work [18].

Clock interpolation requires a calibration before being able to return accurate timestamps. For this purpose, we repeatedly use a busy-wait loop to increment a counter between two clock edges. This gives us the number of steps we can use for the interpolation. We refer to the average number of increments as *interpolation steps*. The time it takes to increment the counter once equals the resolution we are able to recover. It can be approximated by dividing the time difference of two clock edges by the number of interpolation steps. This makes the timer independent from both the internal and the provided resolution.

The measurement with the improved resolution works as follows. We busy wait until we observe a clock edge. At this point, we start with the operation we want to time. After the timed operation has finished, we again busy wait for the next clock edge while incrementing a counter. We assume that the increment operation is a constant time operation, thus allowing us to linearly interpolate the passed time. From the calibration, we know the time of one interpolation step

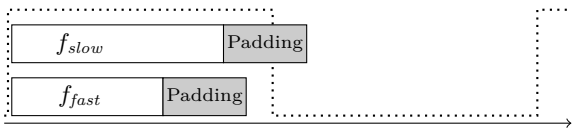

Fig. 1: Edge thresholding: apply padding such that the slow function crosses one more clock edge than the fast function.

which will be a fraction of the provided resolution. Multiplying this time by the number of increments results in the interpolated time. Adding the interpolated time to the measured time increases the timer's resolution again.

Using this method, we recover a highly accurate timestamp. Listing A.1 shows the JavaScript implementation. Table 1 shows the recovered resolution for various values of provided resolution. Even for a timer rounded down to a multiple of 100 ms, we recover a resolution of 15 µs.

Edge thresholding We do not require an exact timestamp in all cases. For many side-channel attacks it is sufficient to distinguish two operations f_{fast} and f_{slow} based on their execution time. We refer to the execution times of the short-running function and long-running function as t_{fast} and t_{slow} respectively.

We devise a new method that we call edge thresholding. This method again relies on the property that we can execute multiple constant-time operations between two edges of the clock. Edge thresholding works as long as the difference in the execution time is larger than the time it takes to execute one such constant-time operation. Figure 1 illustrates the main idea of edge thresholding. Using multiple constant-time operations, we generate a padding after the function we want to measure. The execution time of the padding $t_{padding}$ is included into the measurement, increasing the total execution time by a constant value. The size of the padding depends on the provided resolution and on the execution time of the functions. We choose the padding in such a way that $t_{slow} + t_{padding}$ crosses one more clock edge than $t_{fast} + t_{padding}$, *i.e.*, both functions take a different amount of clock edges.

To choose the correct padding, we start without padding and increase the padding gradually. We align the function start at a clock edge and measure the number of clock edges it takes to execute the short-running and the long-running function. As soon as the long-running function crosses one more clock edge than the short-running function, we have found a working padding. Subsequently, this padding is used for all execution time measurements. Figure 2 shows the results of classifying two functions with an execution time difference of 0.9 µs and a provided resolution of 10 µs. A normal, unaligned measurement is able to classify the two functions only in the case when one of the measurements crosses a clock edge, whereas the edge thresholding method categorizes over 80% of the function calls correctly by their relative execution time. Moreover, there are no false classifications.

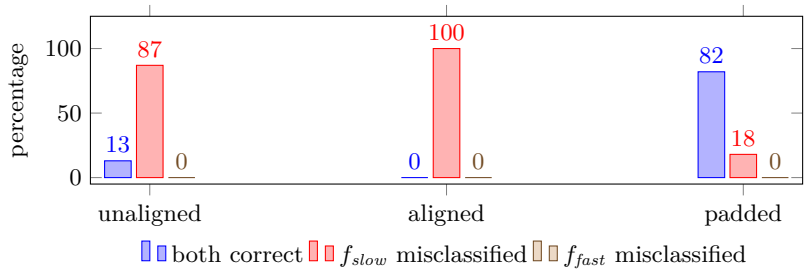

Fig. 2: Results of edge thresholding where the difference between the function's execution time is less then the provided resolution.

3.2 Alternative timing primitives

In cases where the High Resolution Time API [40] is not available, e.g., on Tor browser, we have to resort to different timing primitives, as highlighted by Kohlbrenner et al. [18]. As there is no different high-resolution timer available in JavaScript and we cannot access any native timers, we have to create our own timing sources. In most cases, it is sufficient to have a fast-paced monotonically increasing counter as a timing primitive that is not a real representation of time but an approximation of a highly accurate monotonic timer. While this concept was already presented by Wray in 1992 [44], Lipp et al. [20] recently demonstrated a practical high-resolution timing primitive on ARM using a counting thread. As JavaScript is inherently based on a single threaded event loop with no true concurrency, the timing primitive has to be based either on recurring events or non-JavaScript browser features.

We present several novel methods to construct timing primitives in JavaScript. We refer to them as *free-running timers* and *blocking timers*. Free-running timers do not depend on the JavaScript's event loop and run independently from the remaining code. Blocking timers are based on JavaScript events and are either only usable to recover a high resolution or in combination with web workers. If used in combination with web workers, the timers become free-running timers.

At first, it seems that timing primitives blocking the JavaScript event loop might not be useful at all. The higher the resolution of the timing primitive, the more events are added to the event queue and the less time remains for actual code. However, there are still two constructions that are able to use such primitives. First, these primitives can be used for very accurate interpolation steps when applying either clock interpolation or edge thresholding. Second, it is possible to take advantage of the multithreading support with web workers to run the timing primitive in parallel to the method to time.

Timeouts The first asynchronous feature dating back to the introduction of JavaScript is the WindowTimers API. Specifically the `setTimeout` and `setInterval` functions allow scheduling a timer-based callback. The time is specified in a mil-

lisecond resolution. After specifying the timeout, the browser keeps track of the timer and calls the callback as soon as the timer has expired.

A concurrent timer-based callback allows us to simulate a counting thread. We create a callback function that increments a global counter and schedules itself again using the `setTimeout` function. This method has also been used in a concurrent work [18]. Although the minimal supported timeout is 0, the real timeout is usually larger. The HTML5 specification defines a timeout of at least 4 ms for nested timers, *i.e.*, specifying the timeout from within the callback function has a delay of at least 4 ms [42]. This limitation also applies to timeouts specified by the `setInterval` function.

Most browsers comply to the HTML5 specification and treat all timeouts below 4 ms as 4 ms. In Firefox, the minimum timeout is determined by the value of the flag `dom.min_timeout_value` which defaults to 4 ms as well. Note that the timeout only has such a high frequency if it is run in an active tab. Background tasks do not allow such high frequencies.

Microsoft implemented another timeout function in their browsers which is not standardized. The `setImmediate` function behaves similarly to the `setTimeout` function with a timeout of 0. The function is not limited to 4 ms and allows to build a high-resolution counting thread. A counting thread using this function results in a resolution of up to 50 µs which is three orders of magnitude higher than the `setTimeout` method.

Message passing By default, the browser enforces a same-origin policy, *i.e.*, scripts are not allowed to access web page data from a page that is served from a different domain. JavaScript provides a secure mechanism to circumvent the same-origin policy and to allow cross-origin communication. Scripts can install message listeners to receive message events from cross-origin scripts. A script from a different origin is allowed to post messages to a listener.

Despite the intended use for cross-origin communication, we can use this mechanism within one script as well. The message listener is not limited to messages sent from cross-origin scripts. Neither is there any limitation for the target of a posted message. Adding checks whether a message should be handled is left to the JavaScript developer. According to the HTML standard, posted messages are added to the event queue, *i.e.*, the message will be handled after any pending event is handled. This behavior leads to a nearly immediate execution of the installed message handler. A counting thread using the `postMessage` functions achieves a resolution of up to 35 µs. An implementation is shown in Listing A.2.

To obtain a free-running timing primitive, we have to move the message posting into separate web workers. This appears to be a straightforward task. However, there are certain limitations for web workers. Web workers cannot post messages to other web workers (including themselves). They can only post messages to the main thread and web workers they spawn, so called sub workers. Posting messages to the main thread again blocks the main thread's event loop, leaving sub web workers as the only viable option. Listing A.3 shows a sample implementation using one worker and one sub worker. The worker can communi-

cate with the main thread and the sub worker. If the worker receives a message from the main thread, it sends back its current counter value. Otherwise, the worker continuously "requests" the current counter value from the sub worker. The sub worker increments the counter on each request and sends the current value back to the worker. The resulting resolution is even higher than with the blocking version of the method. On Tor browser, the achieved resolution is up to 15 µs, which is 4 orders of magnitude higher than the resolution of the native timer.

An alternative to sub workers are broadcast channels. Broadcast channels allow the communication between different sources from the same origin. A broadcast channel is identified by its name. In order to subscribe to a channel, a worker can create a `BroadcastChannel` object with the same name as an existing channel. A message that is posted to the broadcast channel is received by all other clients subscribed to this broadcast channel. We can build a construct that is similar to the sub worker scenario using two web workers. The web workers broadcast a message in their broadcast receiver to send the counter value back and forth. One of the web workers also responds to messages from the main thread to return the current counter value. With a resolution of up to 55 µs, this method is still almost as fast as the worker thread variant.

Message Channel The Channel Messaging API provides bi-directional pipes to connect two clients. The endpoints of the pipe are called ports, and every port can both send and receive data. A message channel can be used in a similar way as cross-origin message passing. Listing A.4 shows a simple blocking counting thread using a message channel.

As with the cross-origin message passing method, we can also adapt this code to work inside a web worker yielding a free-running timing primitive. Listing A.5 shows the implementation for web workers. The resolution for the free-running message channel method is up to 30 µs, which is lower compared to the cross-origin communication method. However, it is currently the only method that works across browsers and has a resolution in the order of microseconds.

CSS animations With CSS version 3, the support for animations [39] was added. These animations are independent of JavaScript and are rendered by the browser. Users can specify keyframes and attributes that will then be animated without any further user interaction.

We demonstrate a new method that uses CSS animations to build a timing primitive. A different method using CSS animations has been used in a concurrent work [18]. We define an animation that changes the width of an element from 0 px to 1 000 000 px within 1 s. Theoretically, if all animation steps are calculated, the current width is incremented every microsecond. However, browsers limit the CSS animations to 60 fps, *i.e.*, the resolution of our timing primitive is 16 ms in the best case. Indeed, most monitors have a maximum refresh rate of 60 Hz, *i.e.*, they cannot display more than 60 fps. Thus, a higher frame rate would only waste resources without any benefit. To get the current

timestamp, we retrieve the current width of the element. In JavaScript, we can get the current width of the element using `window.getComputedStyle(elem, null).getPropertyValue("width")`.

SharedArrayBuffer Web workers do not have access to any shared resource. The communication is only possible via messages. If data is passed using a message, either the data is copied, or the ownership of the data is transferred. This design prevents race conditions and locking problems without having to depend on a correct use of locks. Due to the overhead of message passing for high-bandwidth applications, approaches for sharing data between workers are discussed by the ECMAScript committee [27]. An experimental extension for web workers is the `SharedArrayBuffer`. The ownership of such a buffer can be shared among multiple workers, which can access the buffer simultaneously.

A shared resource provides a way to build a real counting thread with a negligible overhead compared to a message passing approach. This already raised concerns with respect to the creation of a high-resolution clock [19]. In this method, one worker continuously increments the value of the buffer without checking for any events on the event queue. The main thread simply reads the current value from the shared buffer and uses it as a high-resolution timestamp.

We implemented a clock with a parallel counting thread using the `SharedArrayBuffer`. An implementation is shown in Listing A.6. The resulting resolution is close to the resolution of the native timestamp counter. On our Intel Core i5 test machine, we achieve a resolution of up to 2 ns using the shared array buffer. This is equivalent to a resolution of only 4 CPU cycles, which is 3 orders of magnitude better than the timestamp provided by `performance.now`.

3.3 Evaluation

We evaluated all methods on an Intel Core i5-6200U machine using the most popular browsers, up to date at the time of writing: Firefox 51, Chrome 53, Edge 38.14393.0.0, and Tor 6.0.4. All tests were run on Ubuntu 16.10, Windows 10, and Mac OS X 10.11.4. Table 1 shows the timing resolution of every method for every browser and operating system combination. We also evaluated our methods using Fuzzyfox [17], the fork of Firefox hardened against timing attacks [18].

The introduction of multithreading in JavaScript opened several possibilities to build a timing primitive that does not rely on any provided timer. By building a counting thread, we are able to get a timer resolution of several microseconds. This is especially alarming for the Tor browser, where the provided timer only has a resolution of 100 ms. Using the demonstrated methods, we can build a reliable timer with a resolution of up to 15 µs. The lower resolution was implemented as a side channel mitigation and is rendered useless when considering the results of the alternative timing primitives.

The best direct timing source we tested is the experimental `SharedArrayBuffer`. The best measurement method we tested is edge thresholding. Both increase the resolution by at least 3 orders of magnitude compared to `performance.now` in all browsers. Countermeasures against timing side-channels using

Table 1: Timing primitive resolutions on various browsers and operating systems.

	Free-running	Firefox 51	Chrome 53	Edge 38	Tor 6.0.4	Fuzzyfox
`performance.now`	✓	5 µs	5 µs	1 µs	100 ms	100 ms
CSS animations	✓	16 ms	16 ms	16 ms	16 ms	125 ms
setTimeout		4 ms	4 ms	2 ms	4 ms	100 ms
setImmediate		–	–	50 µs	–	–
postMessage		45 µs	35 µs	40 µs	40 µs	47 ms
Sub worker	✓	20 µs	–[2]	50 µs	15 µs	–
Broadcast Channel	✓	145 µs	–	–	55 µs	760 µs
MessageChannel		12 µs	55 µs	20 µs	20 µs	45 ms
MessageChannel (W)	✓	75 µs	100 µs	20 µs	30 µs	1120 µs
SharedArrayBuffer	✓	2 ns[3]	15 ns[4]	–	–	2 ns[3]
Interpolation[1]		500 ns	500 ns	350 ns	15 µs	–
Edge thresholding[1]		2 ns	15 ns	10 ns	2 ns	–

fuzzy time have been proposed by Hu et al. [12] and Vattikonda et al. [38]. They suggested to reduce the provided resolution and to randomize the clock edges. However, we can fall back to the constructed timing primitives if this countermeasure is not applied on all implicit clocks.

In a concurrent work, Kohlbrenner et al. [18] proposed Fuzzyfox, a fork of Firefox that uses fuzzy time on both explicit and implicit clocks, and aims to cap all clocks to a resolution of 100 ms. Our evaluation shows that the explicit timer `performance.now` is reduced to 100 ms, and is fuzzed enough that the interpolation and edge thresholding methods do not work to recover a high resolution. Similarly, some of the implicit timers, such as `setTimeout`, `postMessage`, and Message Channel, are also mitigated, with a resolution between 45 ms and 100 ms. However, the Broadcast Channel, Message Channel with web workers, and SharedArrayBuffer still have a fine grained resolution, between 2 ns and 1 ms. It is to be noted that, while these methods stay accurate, the resulting clock is too fuzzy to derive a finer clock with either interpolation or edge thresholding.

[1] Uses `performance.now` for coarse-grained timing information.
[2] Sub workers do not work in Chrome, this is a known issue since 2010 [7].
[3] Currently only available in the nightly version.
[4] It has to be enabled by starting Chrome with `–js-flags=–harmony-sharedarraybuffer –enable-blink-feature=SharedArrayBuffer`.

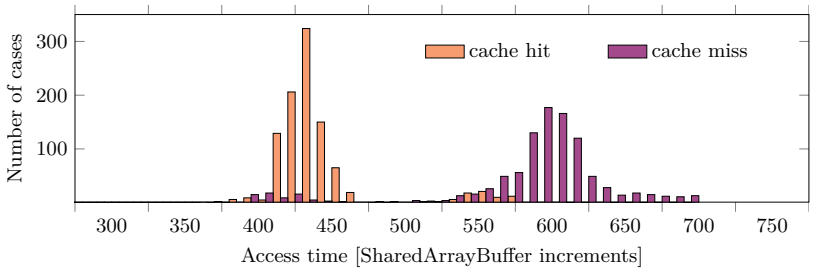

Fig. 3: Histogram for cache hits and cache misses.

4 Reviving and Extending Microarchitectural Attacks

In this section, we demonstrate that with our timing primitives, we are able to revive attacks that were thought mitigated, and build new DRAM-based attacks.

4.1 Reviving Cache Attacks

Oren et al. [28] presented the first microarchitectural side-channel attack running in JavaScript. Their attack was mitigated by decreasing the timer resolution. We verified that the attack indeed does not work anymore on current browser versions. However, we are able to revive cache attacks by using our newly discovered timing sources. Figure 3 shows the timing difference between cache hits and cache misses, measured with the `SharedArrayBuffer` method. The ability to measure this timing difference is the building block of all cache attacks.

4.2 A New DRAM-based Covert Channel

Pessl et al. [32] established that timing differences in memory accesses can be exploited to build a cross-CPU covert channel. We demonstrate that this attack is also possible using JavaScript. In our scenario, the sender is an unprivileged binary inside a VM without a network connection. The receiver is implemented in sandboxed JavaScript running in a browser outside the VM, on the same host.

Overview To communicate, the sender and the receiver agree on a certain bank and row of physical memory. This agreement can be done in advance and is not part of the transmission. The receiver continuously measures the access time to a value located inside the agreed row. For continuous accesses, the value will be cached in the row buffer and the access will be fast, resulting in a low access time. The receiver considers this as a 0. If the sender wants to transmit a 1, it accesses a different row inside the same bank. This access triggers a row conflict, resulting in a replacement of the row buffer content. On the receiver's next access, the request cannot be served from the row buffer but has to be fetched from the DRAM, resulting in a high access time.

Challenges For the sender, we assume that we can run arbitrary unprivileged binary programs inside the VM. We implement the sender in C, which allows us to use the computer's high-resolution timestamp counter. Furthermore, we can flush addresses from the cache using the unprivileged *clflush* instruction. The only limitation on the sender is the absence of physical addresses.

On the receiver side, as the covert channel relies on timing differences that are in the order of tens of nanoseconds, we require a high-resolution timing primitive. We presented in Section 3 different methods to build timing primitives if the provided High Resolution Time API is not accurate enough. However, implementing this side channel in JavaScript poses some problems besides high-resolution timers. First, the DRAM mapping function requires the physical address to compute the physical location, *i.e.*, the row and the bank, inside the DRAM. However, JavaScript does not know the concept of pointers. Therefore, we neither have access to virtual nor physical addresses. Second, we have to ensure that memory accesses will always be served from memory and not the cache, *i.e.*, we have to circumvent the cache. Finally, the noise present on the system might lead to corrupt transfers. We have to be able to detect such bit inversions for reliable communication.

Address selection The DRAM mapping function reverse engineered by Pessl et al. [32] takes a physical address and calculates the corresponding physical memory location. Due to the absence of addresses in JavaScript, we cannot simply use these functions. We have to rely on another side channel to be able to infer address bits in JavaScript.

We exploit the fact that heap memory in JavaScript is allocated on demand, *i.e.*, the browser acquires additional heap memory from the operating system if this is required. These heap pages are internally backed by 2 MB pages, called Transparent Huge Pages (THP). Due to the way virtual memory works, for THPs, the 21 least-significant bits of a virtual and physical address are the same. On many systems, this is already sufficient as input to the DRAM mapping function. This applies to the sender as well, with the advantage that we know the virtual address which we can use immediately without any further actions.

To get the beginning of a THP in JavaScript, we iterate through an array of multiple megabytes while measuring the time it takes to access the array element, similarly to Gruss et al. [9]. As the physical pages for these THPs are also mapped on-demand, a page fault occurs as soon as an allocated THP is accessed for the first time. Such an access takes significantly longer than an access to an already mapped page. Thus, higher timings for memory accesses with a distance of 2 MB indicate the beginning of a THP. At this array index, the 21 least-significant bits of both the virtual and the physical address are 0. Figure 4 shows the timing peaks when accessing a new THP.

Cache circumvention To measure DRAM access times we have to ensure that all our accesses go to the DRAM and not to the cache. In native code, we can rely on the `clflush` instruction. This unprivileged instruction flushes a virtual

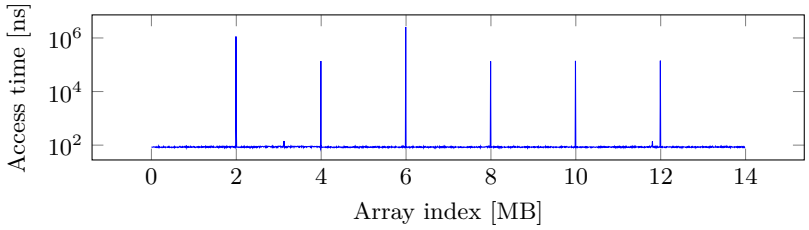

Fig. 4: Access times when iterating through a large array in JavaScript.

address from all cache levels, *i.e.*, the next access to the address is ensured to go to the DRAM.

However, in JavaScript we neither have access to the `clflush` instruction nor does JavaScript provide a function to flush the cache. Thus, we have to resort to cache eviction. Cache eviction is the process of filling the cache with new data until the data we want to flush is evicted from the cache. The straightforward way is to fill a buffer with the size of the last-level cache with data. However, this is not feasible in JavaScript as writing multiple megabytes of data is too slow. Moreover, on modern CPUs, it might not suffice to iteratively write to the buffer as the cache replacement policy is not pseudo-LRU since Ivy Bridge [43].

Gruss et al. [9] demonstrated fast cache eviction strategies for numerous CPUs. They showed that their functions have a success rate of more than 99.75% when implemented in JavaScript. We also rely on these functions to evict the address which we use for measuring the access time.

Transmission To transmit data from inside the VM to the JavaScript, they have to agree on a common bank. It is not necessary to agree on a bank dynamically, it is sufficient to have the bank hardcoded in both programs. The sender and the receiver both choose a different row from this bank. Again, this can be hardcoded, and there is no requirement for an agreement protocol.

On the sender side, the application inside the VM continuously accesses a memory address in its row if it wants to transmit a binary 1. These accesses cause row conflicts with the receiver's row. To send a binary 0, the sender does nothing to not cause any row conflict. On the receiver side, the JavaScript constantly measures the access time to a memory address from its row and evicts the address afterwards. If the sender has accessed its row, the access to the receiver's row results in a row conflict. As a row conflict takes significantly longer than a row hit, the receiver can determine if the sender has accessed its row.

To synchronize sender and receiver, the receiver measures the access time in a higher frequency than the sender is sending. The receiver maintains a constant-size sliding window that moves over all taken measurements. As soon as the majority of the measurements inside the sliding window is the same, one bit is received. The higher the receiver's sampling frequency is, compared to the sender's sending frequency, the lower the probability of wrongly measured bits.

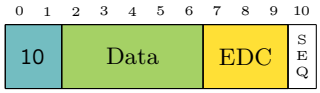

Fig. 5: One packet of the covert channel. It has a 2-bit preamble "10", 5 data bits, 3 bits of error detection code and a 1 bit sequence number.

However, a higher sampling frequency also leads to a slower transmission speed due to the increased amount of redundant data.

Due to different noise sources on the system, we encounter transmission errors. Such noise sources are failed evictions, high DRAM activity of other programs or not being scheduled at all. To have a reliable transmission despite those interferences, we encapsulate the data into packets with sequence numbers and protect each packet with an error detection code as shown in Figure 5. The receiver is then able to detect any transmission error and to discard the packet. The sequence number ensures to keep the data stream synchronized. Thus, transmission errors only result in missing data, but the data stream is still synchronized after transmission errors. To deal with missing data, we can apply high-level error correction as shown by Maurice et al. [24].

Using the `SharedArrayBuffer`, we achieve a transmission rate of 11 bps for a 3 kB file with an error rate of 0% on our Intel Core i5 test machine. The system workload did not influence the transmission, as long as there is at least one core fully available to the covert channel. We optimized the covert channel for reliability and not speed. We expect that it is possible to further increase the transmission rate by using multiple banks to transmit data in parallel. However, the current speed is two orders of magnitude higher than the US government's minimum standard for covert channels [36].

5 Countermeasures

Lowering the timer resolution As a reaction to the JavaScript cache attacks published by Oren et al. [28], browsers reduced the resolution of the high-resolution timer. Nevertheless, we are able to recover a higher resolution from the provided timer, as well as to build our own high-resolution timers.

Fuzzy time Vattikonda et al. [38] suggested the concept of fuzzy time to get rid of high-resolution timers in hypervisors. Instead of rounding the timestamp to achieve a lower resolution, they move the clock edge randomly within one clock cycle. This method prevents the detection of the underlying clock edge and thus makes it impossible to recover the internal resolution. In a concurrent work, Kohlbrenner et al. [18] implemented the fuzzy time concept in Firefox to show that this method is also applicable in JavaScript. The implementation targets explicit clocks as well as implicit clocks. Nonetheless, we found different implicit clocks exceeding the intended resolution of 100 ms.

Shared memory and message passing A proposed mitigation is to introduce thread affinity to the same CPU core for threads with shared memory [19]. This prevents true parallelism and should therefore prevent a real asynchronous timing primitive. However, we showed that even without shared memory we can achieve a resolution of up to 15 µs by using message passing. Enforcing the affinity to one core for all communicating threads would lead to a massive performance degradation and would effectively render the use of web workers useless. A compromise is to increase the latency of message passing which should not affect low- to moderate-bandwidth applications. Compared to Fuzzyfox's delay on the main event queue, this has two advantages. First, the overall usability impact is not as severe as only messages are delayed and not every event. Second, it also prevents the high accuracy of the Message Channel and Broadcast Channel method as the delay is not limited to the main event queue.

6 Conclusion and Outlook

High-resolution timers are a key requirement for side-channel attacks in browsers. As more side-channel attacks in JavaScript have been demonstrated against users' privacy, browser vendors decided to reduce the timer resolution.

In this article, we showed that this attempt to close these vulnerabilities was merely a quick-fix and did not address the underlying issue. We investigated different timing sources in JavaScript and found a number of timing sources with a resolution comparable to `performance.now`. This shows that even removing the interface entirely, would not have any effect. Even worse, an adversary can recover a resolution of the former `performance.now` implementation through measurement methods we proposed. We evaluated our new measurement methods on all major browsers as well as the Tor browser that has applied the highest penalty to the timer resolution. Our results are alarming for all browsers, including the privacy-conscious Tor browser, as we are able to recover a resolution in the order of nanoseconds in all cases. In addition to reviving attacks that were now deemed infeasible, we demonstrated the first DRAM-based side channel in JavaScript. In this side-channel attack, we implemented a covert channel between an unprivileged binary in a VM with no network interface and a JavaScript program in a browser outside the VM, on the same host.

While fuzzy timers can lower the resolution of the provided timer interfaces, we show that applying the same mitigation on all implicit clocks, including the one that are not discovered yet, is a complex task. Thus, we conclude that it is likely that an adversary can obtain sufficiently accurate timestamps when running arbitrary JavaScript code. As microarchitectural attacks are not restricted to JavaScript, we recommend to mitigate them at the system- or hardware-level.

References

1. Alex Christensen: Reduce resolution of performance.now. `https://bugs.webkit.org/show_bug.cgi?id=146531` (2015)

2. Bernstein, D.J.: Cache-Timing Attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf (2004)
3. Boris Zbarsky: Reduce resolution of performance.now. https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab
4. Bortz, A., Boneh, D.: Exposing private information by timing web applications. In: WWW'07 (2007)
5. Bosman, E., Razavi, K., Bos, H., Giuffrida, C.: Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In: S&P'16 (2016)
6. Chromium: window.performance.now does not support sub-millisecond precision on Windows. https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110 (2015)
7. Chromium Bug Tracker: HTML5 nested workers are not supported in chromium. https://bugs.chromium.org/p/chromium/issues/detail?id=31666 (2010), retrieved on October 18, 2016
8. Felten, E.W., Schneider, M.A.: Timing attacks on web privacy. In: CCS'00 (2000)
9. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA'16 (2016)
10. Gullasch, D., Bangerter, E., Krenn, S.: Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In: S&P'11 (2011)
11. Heiderich, M., Niemietz, M., Schuster, F., Holz, T., Schwenk, J.: Scriptless attacks: stealing the pie without touching the sill. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 760–771. ACM (2012)
12. Hu, W.M.: Lattice Scheduling and Covert Channels. In: S&P'92. pp. 52–61 (1992)
13. Jang, D., Jhala, R., Lerner, S., Shacham, H.: An empirical study of privacy-violating information flows in javascript web applications. In: CCS'10 (2010)
14. Jia, Y., Dong, X., Liang, Z., Saxena, P.: I know where you've been: Geo-inference attacks via the browser cache. IEEE Internet Computing 19(1), 44–53 (2015)
15. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: ISCA'14 (2014)
16. Kocher, P.C.: Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: Crypto'96. pp. 104–113 (1996)
17. Kohlbrenner, D., Shacham, H.: Fuzzyfox. https://github.com/dkohlbre/gecko-dev/tree/fuzzyfox (2016), retrieved on January 23, 2017
18. Kohlbrenner, D., Shacham, H.: Trusted browsers for uncertain times. In: USENIX Security Symposium (2016)
19. Lars T Hansen: Shared memory: Side-channel information leaks. https://github.com/tc39/ecmascript_sharedmem/blob/master/issues/TimingAttack.md (2016)
20. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium (2016)
21. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-Level Cache Side-Channel Attacks are Practical. In: S&P'15 (2015)
22. Martin, R., Demme, J., Sethumadhavan, S.: TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In: Proceedings of the 39th International Symposium on Computer Architecture (ISCA'12) (2012)
23. Maurice, C., Neumann, C., Heen, O., Francillon, A.: C5: Cross-Cores Cache Covert Channel. In: DIMVA'15 (2015)

24. Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Alberto Boano, C., Mangard, S., Römer, K.: Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS'17 (2017), to appear
25. Mike Perry: Bug 1517: Reduce precision of time for Javascript. https://gitweb.torproject.org/user/mikeperry/tor-browser.git/commit/?h=bug1517 (2015)
26. Mozilla Developer Network: Concurrency model and Event Loop. https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop (2016)
27. Mozilla Inc.: Ecmascript shared memory and atomics. http://tc39.github.io/ecmascript_sharedmem/shmem.html (2016)
28. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS'15 (2015)
29. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA 2006 (2006)
30. Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive, Report 2002/169 (2002)
31. Percival, C.: Cache missing for fun and profit. In: Proceedings of BSDCan (2005)
32. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium (2016)
33. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: CCS'09 (2009)
34. Seaborn, M.: Comment on ecmascript shared memory and atomics. https://github.com/tc39/ecmascript_sharedmem/issues/1#issuecomment-144171031 (2015)
35. Stone, P.: Pixel perfect timing attacks with html5. Context Information Security (White Paper) (2013)
36. U.S. Department of Defense: Trusted computing system evaluation "the orange book". Technical Report 5200.28-STD (1985)
37. Van Goethem, T., Joosen, W., Nikiforakis, N.: The clock is still ticking: Timing attacks in the modern web. In: CCS'15 (2015)
38. Vattikonda, B.C., Das, S., Shacham, H.: Eliminating fine grained timers in xen. In: CCSW'11 (2011)
39. W3C: CSS Animations. https://www.w3.org/TR/css3-animations/ (2016)
40. W3C: High Resolution Time Level 2. https://www.w3.org/TR/hr-time/ (2016)
41. Weinberg, Z., Chen, E.Y., Jayaraman, P.R., Jackson, C.: I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In: S&P'11 (2011)
42. WHATWG: HTML Living Standard – Timers. https://html.spec.whatwg.org/multipage/webappapis.html#timers (2016), retrieved on October 18, 2016
43. Wong, H.: Intel Ivy Bridge Cache Replacement Policy. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/, retrieved on October 18, 2016
44. Wray, J.C.: An analysis of covert timing channels. Journal of Computer Security 1(3-4), 219–232 (1992)
45. Wu, Z., Xu, Z., Wang, H.: Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. IEEE/ACM Transactions on Networking (2014)
46. Xiao, J., Xu, Z., Huang, H., Wang, H.: A covert channel construction in a virtualized environment. In: CCS'12 (2012)

47. Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., Schlichting, R.: An exploration of L2 cache covert channels in virtualized environments. In: CCSW'11 (2011)
48. Yarom, Y., Falkner, K.: Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium (2014)

A JavaScript Code

```
function calibrate()
{
  var counter = 0, next;
  for(var i = 0; i < 10; i++)
  {
      next = wait_edge();
      counter += count_edge();
  }
  next = wait_edge();
  return (wait_edge() - next) /
    (counter / 10.0);
}

function measure(fnc)
{
  var start = wait_edge();
  fnc();
  var count = count_edge();
  return (performance.now()-start)
      - count * calibrate();
}
```

a: Clock interpolation.

```
function wait_edge()
{
  var next, last =
      performance.now();
  while((next =
      performance.now())
      == last) {}
  return next;
}

function count_edge()
{
  var last = performance.
      now(), count = 0;
  while(performance.now()
      == last) count++;
  return count;
}
```

b: Helper functions.

Listing A.1: Clock interpolation: `calibrate` returns the time one increment takes, `measure` uses interpolation to measure the execution time of `fnc`

```
var count = 0;

function counter()
{
    count++;
    window.postMessage(null, window.location);
}
window.addEventListener("message", counter);
window.postMessage(null, window.location);
```

Listing A.2: Abusing cross-origin communication to build a counting thread.

```
var ts = new
  Worker('subworker.js');
ts.postMessage(0);

function counter(event)
{
  timestamp = event.data;
}
ts.addEventListener("message",
     counter);
[...]

// get timestamp
ts.postMessage(0);
```

a: Timing measurement example.

```
var count = 0;

onmessage = function(event)
{
    count++;
    postMessage(count);
}
```

b: subworker2.js

```
var sub = new
  Worker("subworker2.js");
sub.postMessage(0);

var count = 0;

sub.onmessage = msg;
onmessage = msg;

function msg(event)
{
    if(event.data != 0)
    {
        count = event.data;
        sub.postMessage(0);
    }
    else
      self.postMessage(
          count);
}
```

c: subworker.js

Listing A.3: Message passing with web workers to get a free-running timer.

```
var count = 0, channel = null;
function handleMessage(e)
{
    count++;
    channel.port2.postMessage(0);
}

channel = new MessageChannel();
channel.port1.onmessage = handleMessage;
channel.port2.postMessage(0);
```

Listing A.4: A blocking timing primitive using a message channel.

```
var worker = new
  Worker("mcworker.js");
var main_channel = new
     MessageChannel();
var side_channel = new
     MessageChannel();

function handleMessage(e)
{
    timestamp = e.data;
}

main_channel.port2.onmessage =
     handleMessage;
worker.postMessage(0,
  [ main_channel.port1,
       side_channel.port1,
       side_channel.port2 ]);
[...]

// get timestamp
main_channel.port2.postMessage
     (0);
```

a: Timing measurement example.

```
var main_port, port1, port2,
     count = 0;

self.onmessage = function(
     event)
{
    main_port = event.ports
         [0];
    port1 = event.ports[1];
    port2 = event.ports[2];
    main_port.onmessage =
      function()
      {
        main_port.postMessage(
             count);
      };
    port1.onmessage =
      function()
      {
        count++;
        port2.postMessage(0);
      };
    port2.postMessage(count);
};
```

b: mcworker.js

Listing A.5: Message passing with web workers to get a free-running timer.

```
var buffer = new
     SharedArrayBuffer(16);
var counter = new
  Worker("counter.js");
counter.postMessage([buffer],
  [buffer]);
var arr = new
  Uint32Array(buffer);
[...]

timestamp = arr[0];
```

a: Timing measurement example.

```
self.onmessage = function(
     event)
{
  var [buffer] = event.data;
  var arr = new
    Uint32Array(buffer);
  while(1)
  {
    arr[0]++;
  }
}
```

b: counter.js

Listing A.6: Parallel counting thread without additional overhead.