JSKernel: Fortifying JavaScript against Web Concurrency Attacks via a Kernel-like Structure

Zhanhao Chen† and Yinzhi Cao‡
lhdxczh@gmail.com, ycao43@jhu.edu
† Palo Alto Networks ‡ Johns Hopkins University

***Abstract*—As portals to the Internet, web browsers constitute prominent targets for attacks. Existing defenses that redefine web APIs typically capture information related to a single JavaScript function. Thus, they fail to defend against the so-called *web concurrency attacks* that use multiple interleaved functions to trigger a browser vulnerability.**

In this paper, we propose JSKERNEL, the first generic framework that introduces a kernel concept into JavaScript to defend against web concurrency attacks. The JavaScript kernel, inspired from operating system concepts, enforces the execution order of JavaScript events and threads to fortify security.

We implement a prototype of JSKERNEL deployable as add-on extensions to three widely used web browsers, namely Google Chrome, Mozilla Firefox, and Microsoft Edge. These open-source extensions are available at (https://github.com/jskernel2019/jskernel) along with a usability demo at (https://jskernel2019.github.io/). Our evaluation shows the prototype to be robust to web concurrency attacks, fast, and backward compatible with legacy websites.

I. INTRODUCTION

Web browsers typically form the front line across defenders and adversaries resulting from the direct access to untrusted web content on the Internet. The problem is exacerbated by the rapid evolution of the World Wide Web along with the continuous introduction of multiple new features (e.g., Web Workers), and the corresponding new vulnerabilities.

A popular direction in securing browsers is that of attack surface reduction that aims to constrain specific web functionalities. Such an approach dates to the early days of the web where the advocacy was for disabling JavaScript to improve security. More recently, Snyder et al. [1] also proposed to disable certain high-risk APIs, such as Web Workers, to fortify security. However, as the web continues to evolve with new feature-based functionalities, the disabling approach is not always realistic. For example, Web Workers are widely used for background computations in popular websites (e.g., Google Maps and Overleaf), and disabling workers would detrimentally affect the user experience.

Another widely-pursued direction to secure web browsers is to precisely detect the triggering condition of web attacks and stop them before they happen. Such an approach dates back to the seminal BrowserShield [2] work where Reis et al. proposed to rewrite JavaScript in a proxy to enforce security policies. Such methods of rewriting JavaScript in a proxy have gradually evolved to the redefinition of JavaScript APIs via browser add-ons. For example, more recently, JavaScript Zero [3] redefines JavaScript APIs in browser add-ons to prevent side-channel attacks. Similarly, JavaScript API redefinition is widely utilized in many commercial products [4], [5], and approaches such as Canvas Defender [4] redefine canvas API to introduce noise and defend against canvas fingerprinting. However, all these existing API redefinition works rely on an implicit assumption that the triggering condition of web attacks needs information only from the redefined API. For example, BrowserShield enforces simple security policies that check the length of API calls to prevent buffer overflow. Analogously, JavaScript Zero redefines $performance.now$, a modern, fine-grained clock API, to reduce its precision.

However, the above assumption no longer holds in modern web browsers that include full-fledged, event-driven operating system architectures with multiple threads. Specifically, we show that the triggering condition of many modern web attacks needs information, i.e., the invocation sequence, of multiple JavaScript functions spread across different web threads. In this paper, we define these attacks as *Web Concurrency Attacks*. These attacks cannot be prevented using simple API re-definitions proposed in prior work.

The most prominent web concurrency attack is probably the web implicit clock [6] used in side-channel attacks, that can also transpire in many prior attacks [7], [8], [9], [10], [11], [12]. In particular, the triggering condition of an implicit clock is the interleaved invocations of two JavaScript functions: one is the implicit clock API, such as the $onmessage$ callback of $postMessage$ function, and the other is the measurement target API. The number of invoked $onmessage$ callbacks is used to infer the duration of the target API invocation. Prior approaches that redefine individual APIs (such as JavaScript Zero) cannot prevent web implicit clocks. This is because the attack is related to the invocation sequence of multiple JavaScript functions, which cannot be captured by the security policy using information from individual functions.

Web concurrency attacks also extend beyond implicit clocks in timing side-channel attacks. Consider the use-after-free vulnerability [13] documented in the Common Vulnerabilities and Exposures (CVE) Database. The triggering condition of this vulnerability involves three JavaScript functions, namely the fetch initiating a web request, a false termination of the worker, and an abort signal to the fetch. Specifically, the fetch has to occur in the worker, the worker has to be falsely terminated due to a bug, and then the main thread needs to send an abort signal to the terminated worker to finally trigger the use-after-free vulnerability. Again, JavaScript Zero cannot

capture the correlation among these three functions. JavaScript Zero only replaces the native implementation of workers with a nonparallel version like a polyfill, which sacrifices true parallelism of web browsers.

In this paper, we propose JSKERNEL, the first framework that introduces a kernel concept with the capability of capturing concurrency information of different threads into JavaScript to defend against web concurrency attacks. The core kernel concept, of an additional layer with a higher privilege in between the browser and the website JavaScript, is inspired by the operating system (OS) kernel analogy. The *key* idea is that the JavaScript kernel, similar to an OS kernel, manages all the JavaScript threads and schedules events in each thread following a certain security policy thus preventing a web concurrency attack. The policy for the implicit clock example can be deterministic [14] or fuzzy [6] scheduling; the one for the aforementioned use-after-free vulnerability is a manual specification that explicitly closes the already-terminated worker after the fetch and before the abort signal.

We have implemented a prototype of JSKERNEL as extensions to three major browsers, i.e., Firefox, Chrome and Edge. Our implementation is open-source and available at the following repository (https://github.com/jskernel2019/jskernel). We also provide an demo of the attack at https://jskernel2019.github.io/. Our design and implementation of the JSKERNEL prototype follows the following principles:

- Browser-agnostic. We design JSKERNEL to be deployable at any existing web browser. Specifically, we design JSKERNEL with a piece of thin extension code for bootstrapping and a portal kernel runnable at any existing browser.
- Backward compatibility. We design JSKERNEL to be backward compatible with legacy websites. For example, in timing APIs, the execution sequence of asynchronous event enforced by JSKERNEL is one out of many possibilities, thus being compatible with legacy websites.
- High performance. JSKERNEL incurs insignificant overhead compared with legacy execution of web applications. Our evaluation shows that JSKERNEL only incurs 0.30% median overhead on the Dromaeo benchmark [15].

II. OVERVIEW

In this section, we provide an overview of JSKERNEL. We start by describing the web concurrency attack in Section II-A and present examples of security policies needed for preventing web concurrency attacks in Section II-B.

A. Threat Model: Web Concurrency Attack

Web concurrency attack is defined as a web-level attack triggered by a specific invocation sequence of two or more JavaScript built-in functions, e.g., system APIs and callbacks, with certain parameters and possibly located in multiple threads. The consequence of web concurrency attack varies, where the manifestation could be a privacy leak of cross-origin information or an exploit of a low-level vulnerability. The common thread among such attacks is that the triggering condition requires a particular invocation sequence of JavaScript functions.

```
1  // worker.js:
2  function(){
3    for (var i = 0; i < BIG_NUMBER; i ++)
4      postMessage(i);
5  }
6  // Main Script:
7  <style>.f { filter: url(#morphology) } </style>
8  <img src="http://cross-origin.org/input.png" id="e">
9  <svg>
10   <filter id="morphology">
11   <feMorphology operator="erode" radius="30">
12   </filter>
13 </svg>
14 <script>
15 worker = new Worker("worker.js");
16 startTime = performance.now();
17 worker.onmessage = function (event) {
18   count = event.data;
19   if (event.data == NUM) {
20    tick = (performance.now() - startTime)/NUM;
21    callback = function(){
22      asyncTimerDuration = tick * (count - NUM);
23    }
24    document.getElementById("e").classList.toggle('f');
25    requestAnimationFrame(callback);
26   }
27 }
28 </script>
```

Listing 1: Web Concurrency Attack Example 1: A JavaScript worker uses the callback function of postMessage as an implicit clock.

Note that web concurrency attacks differ from low-level concurrency attacks [16]. Low-level concurrency attacks are mostly caused by a race condition, which could then lead to, for example, a privilege escalation from the user space to the kernel. The cause of web concurrency attacks, if targeting a low-level vulnerability, is that a particular invocation sequence of JavaScript functions across multiple threads will result in control and data flows at the low level such that the vulnerability can be triggered.

Next, we present two examples to illustrate web concurrency attacks. We start from an implicit clock example in Section II-A1 that measures unknown information, e.g., loading time of a cross-origin resource in timing attacks, and then describe a low-level use-after-free vulnerability in Section II-A2.

1) Attack Example 1—An implicit web clock: We first describe a timing attack with an implicit clock in Listing 1 to illustrate web concurrency attacks. Consider the general pattern of a timing attack with an implicit clock. An adversary measures a secret, e.g., the operational time of an SVG filter (Lines 7–13), using the number of invocations of an implicit clock API, e.g., *onmessage* events (Lines 17–27) triggered by the *postMessage* call (Line 4) in a JavaScript worker (Lines 2–5). As the operational time of the SVG filter differs based on the image contents, an adversary can infer the image contents based on prior works [9]. The use of an implicit clock belongs to a web concurrency attack because of the interleaved invocation of the secret and the implicit clock API.

The attack transpires in two stages covering the measurement of (i) a tick and (ii) a secret. First, similar to the clock

```
1  // worker.js
2   var abortCtl0 = new AbortController();
3   var abortSig0 = abortCtl0.signal;
4   setInterval(function (e) {
5    fetch("./fetchedfile0.html", {signal:abortSig0}).then(
         function(e){...}).catch(function(e) {...});
6   }, 32);
7  // Main Script
8  <script type="text/javascript">
9    var worker = new Worker("worker.js");
10   setTimeout(function(){location.reload();},300);
11 </script>
```

Listing 2: Web Concurrency Attack Example 2: A use-after-free vulnerability triggered by a complex function invocation sequence.

edge attack [6], the adversary needs to measure the length of a clock tick, i.e., the invocation time of *onmessage* event in the current browser that includes an $i++$ (Line 3) and a message passing from the worker to the main thread (Line 4). Particularly, the adversary executes the operation multiple times and then divides the overall duration by the number of executions to obtain one tick's length (Line 20). Second, the adversary invokes the target, i.e., an erode operation, and measures the number of *onmessage* events from the worker thread. Then, the duration of the operation will be the tick length multiplied by the event number (i.e., *asyncTimerDuration* in Line 22).

2) Attack Example 2—A use-after-free vulnerability: A web concurrency attack targeting a use-after-free vulnerability (CVE-2018-5092) is depicted in Listing 2. The vulnerability is a use-after-free where the browser code sends an abort signal to a fetch request that has already been freed due to a false worker termination.The triggering code, simplified from Bugzilla [13] while preserving its functionality, first registers a fetch in the worker thread at Line 4, causes the false termination in the fetch request at Line 5, and then triggers the abort signal by closing the main thread at Line 10. The exploitation of this use-after-free vulnerability is a web concurrency attack because of the strict invocation sequence of these JavaScript APIs across the worker and main threads.

B. Security Policy

We illustrate some example security policies adopted by JSKERNEL to defend against web concurrency attacks. A security policy in JSKERNEL, represented in a JSON format and specifies the corresponding functions to be invoked for a user-space, i.e., a website JavaScript, function call in either the main or the worker thread. A security policy has access to kernel objects, such as the event queue described subsequently in Section III-C, such that it can schedule events in a specific sequence for defense.

We present two examples of security policies in defending the aforementioned web concurrency attacks in Section II-A and then present how to specify policies in general.

1) A Deterministic Scheduling Policy against Implicit Clocks: We outline a security policy implementing a deterministic scheduling in Listing 3. The policy arranges all the events, such as *onmessage*, in a deterministic order. Particularly, the policy creates a pending *onmessage* for each callback of

```
1  policy_deterministic = {
2    worker: { // policy for worker thread
3      JSKernel_WorkerPostMessage: (callback) => {
4        var pendingOnMessage = new Event(callback, [], "
             pending");
5        var expectedTime = predictOnMessage();
6        event_queue.push(pendingOnMessage, expectedTime);
7      }
8    },
9    main: { // policy for main thread
10     JSKernel_WorkerPrototypeOnmessage: (e) => {
11       event_queue.lookup(e.command).status = "
             confirmed";
12   }
13 }
```

Listing 3: A Security Policy of Deterministic Scheduling to Defend against Attack Example One

```
1  policy_cve-2018-5092 = {
2    main: { // policy for main thread
3      JSKernel_WorkerPrototypeOnmessage: (e) => {
4        if (e.command == "pendingChildFetch") {
5          //inform the worker thread
6          this.postSysMsg("confirmFetch", e.id);
7          // close the worker if it is freed after fetch
8          var cleanWorker = new Event(() => {
9            if (!this.alive) this.terminate()
10         });
11         event_queue.push(cleanWorker,
               expectedTimeforFetch);
12       }
13     }
14   },
15   worker: { // policy for worker thread
16     JSKernel_Fetch: (url) => {
17       //inform main thread
18       var kernelFetch = new Event(legacy_fetch, [url], "
             pending");
19       postSysMsg("pendingChildFetch", kernelFetch.id);
20       event_queue.push(kernelFetch,
             expectedTimeforConfirm);
21       return kernelFetch.stub;
22     },
23     JSKernel_onmessage: (e) => {
24       if (e.command == "confirmFetch")
25         event_queue.lookup(e.command).status = "
               confirmed";
26     }
27   }
28 }
```

Listing 4: A Manually-specified Scheduling Policy to Defend against CVE-2018-5092

postMessage (Line 4), predicts a deterministic time (Line 5), and then pushes the pending event into an event queue (Line 6). Once the real *onmessage* event happens, the policy changes the status of the pending event to "confirmed" (Line 10) and waits until its turn to execute the event.

Such a policy can defend against the aforementioned implicit clock. Due to the deterministic scheduling policy, all the *onmessage* and *callback* invocations are arranged deterministically in the time axis. There, the number of *onmessage* invocations in between the starting and ending of *callback* in Listing 1 is deterministic, i.e., *count* (Line 24) is fixed, and so is *asyncTimerDuration*.

2) A Scheduling Policy Preventing Triggering Condition of CVE-2018-5092: In this part, we show a scheduling policy in Listing 4 to defend against the aforementioned use-after-free vulnerability. When a worker thread initiates a *fetch* call, the policy asks the kernel code at the worker thread to send a "pendingChildFetch" message to the main thread

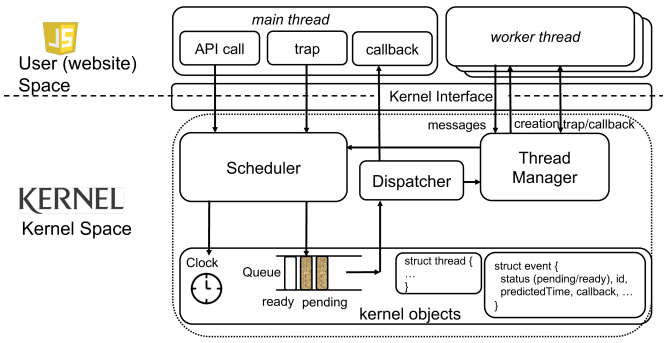

Figure 1: JSKERNEL Architecture

(Line 19), which confirms the receipt of the message via a "confirmFetch" and also creates a "cleanWorker" event to check the liveness of the worker thread and prevent any abort signal from the main thread by closing the worker thread (Lines 6–11). This policy can defend against attacks targeting CVE-2018-5092 because the main thread will be aware of the termination of a worker and thus avoid sending an abort signal to the *fetch* function.

3) Policy Specification: Currently, there are two types of policies, general and specific, which are allowed in JSKERNEL. The aforementioned examples belong to these two types. The policy in Listing 3 is a general one that defends against timing attacks. The policy in Listing 4 is specific to CVE-2018-5092. The writing of specific policies is manual and requires the understanding of the vulnerability triggering condition. We utilize CVE-2018-5092 to explain how a policy is written. An expert reads and understands the exploit code in Bugzilla to extract two critical triggering conditions: (i) a fetch call that causes a false termination in one thread and (ii) a reload that causes another termination in another. Subsequently, the expert writes the policy in Listing 4 to model the interplay between these two triggering conditions.

III. JSKERNEL DESIGN

We now outline the design of JSKERNEL.

A. Overall Architecture

The architecture of JSKERNEL, similar to an OS kernel, has two areas classified per their privilege levels, i.e., the kernel and the website (or called the user space as borrowed from OS). Our kernel has four major components: a storage place of kernel objects, a scheduler, a dispatcher, and a thread manager. The storage place holds all the kernel-related objects, such as a clock, an event queue, JavaScript objects (undefined and Worker) used in the kernel along with event and thread structures. The scheduler handles all the user-to-kernel communication and places events into the event queue, e.g., a manually specified policy or a deterministic scheduling. The dispatcher, essentially an event loop, fetches and invokes all the ready events following the time sequence determined by the scheduler. The thread manager is responsible for creating kernel threads, which will spawn user threads. Thread communication is also handled by the thread manager.

```
1  // kernelworker.js:
2  function(){
3    kernelWorkerInterface = (function(){
4      kernelPostMessage = postMessage;
5      postMessage = function (e) {...};
6      return {src:...};
7    })();
8  importScripts(kernelWorkerInterface.src); //
       kernelWorkerInterface.src="worker.js"
9  }
10 // Kernel script in main thread:
11 var kernelInterface = (function(){
12    kernelWorker = Worker;
13    constructWorker = function (userWorker) {...};
14    registerMsg = function () {...};
15    requestAnimationFrame = function () {...};
16    return {constructWorker, registerMsg, ...};
17 })();
18 var worker_handler = {
19   set: (obj, prop, val) => {
20     if (prop=="onmessage")
21       kernelInterface.registerMsg(...);
22   },// kernel trap
23   construct: (obj, prop) => {
24       var myworker = {name:prop[0]};
25       kernelInterface.constructWorker(myworker);
26       return new Proxy(myworker, worker_handler);
27   }
28 }
29 var Worker=new Proxy(Function, worker_handler);
30 // original attack scripts
31 var worker = new Worker("worker.js");
32 worker.onmessage = function (event) {...};
```

Listing 5: Kernel Interface Code for User-Kernel Communication

B. Kernel Interface

The kernel interface provides a set of APIs for the user space, i.e., the website JavaScript ops. When the user-level code calls the corresponding APIs, the kernel code will be invoked instead of the browser native code. Listing 5 provides an example: *Worker* (Line 29) and *postMessage* (Line 5) are user-space APIs. When the user code tries to create a new worker thread, our kernel code will first create a kernel worker thread (Line 2–9). Then, the kernel code in main thread will communicate the user creation request to the kernel worker, which will then subsequently import the user worker (Line 10) under a similar environment like the main thread.

We now categorize the interfaces into two types of communications: user→kernel and kernel→user.

1) User→Kernel Communication: The user space may invoke a kernel space function via the following three methods:

- Kernel API calls. Kernel API calls are that a user-space script calls an API, such as setTimeout and postMessage, which are redefined by the kernel space. This redefined API will call the corresponding functions in the kernel space.
- Kernel Traps. Kernel traps result when a user-space script access an object property, the access will be automatically trapped to the kernel. We implement traps via the setter function provided by JavaScript. For example, the code, i.e., *Object.defineProperty (this, 'onmessage', { set: function(e) { ... } });* defines a setter function for onmessage such that when the user space accesses onmessage, the access will be trapped to the setter function in the kernel space.

- User-space Stub. The stub provides a user-space object, which calls a corresponding kernel space function. The Worker object (Line 27 in Listing 5) is such an example stub—all the accesses to the Worker object will be redirected to the worker_handler in the user space. Then, the $worker\_handler$ invokes methods provided by the kernel. Take the new operation for example. When the website JavaScript creates a $Worker$ object, i.e., a Proxy instance, the Proxy will invoke the constructor in $worker\_handler$, which then calls $constructWorker$ method provided by the kernel interface.

Note that the user-space script is free to redefine any API or objects provided in the kernel interface. There could be two reasons. First, as a legitimate case, the user-space script may obtain the old definition and call the old one in the newly defined function. For example, requestAnimationFrame is obtained as a backup copy and then redefined in youtube.com. In such cases, the user-space script obtains our kernel interface API but thinks that it is the original definition. The user-space function will eventually call our kernel interface API in their backup copy invocation. Second, as an adversarial case, an attacker may try to bypass our kernel interface by redefinition. In such an example, although the attacker can bypass our kernel, she cannot launch any attacks either because timing-related objects are encapsulated in the kernel, which the attacker cannot access. The attacker cannot use Object.defineProperty to redefine setter functions of critical properties like onmessage either, because such properties are not configurable.

2) Kernel→User Communication: When a user-space script requests the kernel to finish a task, the kernel needs to communicate with the user space after fulfilling the tasks—such communication is done via a callback function. Usually, the callback function is passed to the kernel when the user-space script requests the task. For example, the function passed to onmessage at Line 30 of Listing 5 is a callback. When the kernel decides to invoke the callback, i.e., via the dispatcher component, the kernel needs to prepare the correct execution context and the arguments. For example, consider the onload event in the document object model (DOM). The $onload$ event of a DOM element needs to be executed with the $this$ object as the element. Therefore, the kernel first binds the callback function with the correct $this$ object, and then applies the callback with the arguments returned from the browser under the correct execution context.

C. Kernel Objects

We now introduce two key types of kernel objects.

1) Event Queue: An event queue arranges all the events, i.e., items in the queue, based on the predicted time. The event queue supports regular queue APIs. For example, a pop API returns the event with the smallest $predictedTime$ and removes it from the queue. Similarly, a top API returns the same event but still keeps it in the queue. Next, the push API inserts an event into the queue and puts it along with other events based on their $predictedTime$ value. Lastly, the remove API removes an event from the queue regardless of its $predictedTime$.

2) Clock: A clock in JSKERNEL is simply a counter that ticks based on certain information, which could be a physical clock tick or specific API calls. A clock object provides two APIs in the kernel space: ticking and displaying. First, the ticking API allows the clock to tick either by or to a certain value. Second, the displaying API allows the clock to return the current time, when a kernel function asks for it. For example, both $performance.now$ and the callback function of $requestAnimationFrame$ need to use this API to query current time.

D. Scheduler and Dispatcher

We now detail event scheduling and cancelling in JSKERNEL.

1) Event Scheduling: The JSKERNEL schedules an event via two steps: registration and confirmation. In the registration step, the scheduler pushes a pending event with a predicted time into the event queue. In the confirmation step, the scheduler confirms the arguments, $this$ object, and sometimes callback for the pending event to result in the change of its status to ready.

First, the registration stage, typically initiated by the user-space scripts calling a kernel API such as $requestAnimationFrame$, prepares an event object with its predicted invocation time and callback function. Specifically, the scheduler creates an empty, pending event object, and predicts the time for the event based on the current time, i.e., querying the clock, and the registration type—the prediction depends on the detailed scheduling algorithm, such as determinism and fuzzy time. Next, the scheduler prepares the callback function for the event. In some cases like $setTimeout$, the callback is unique; in some cases, such as image loading, the callback varies on external factors—an $onload$ callback is fired if the image is available, and otherwise an $onerror$. The scheduler puts all the possible callbacks in the event object and pushes the event object into the event queue. After that, the scheduler registers a kernel callback function with the browser under the original registration type to trigger the confirmation stage with the created event object.

Second, when the browser initiates the kernel callback function, the confirmation stage is automatically triggered. In this stage, the scheduler needs to put the event arguments and $this$ object into the event object. If the event object has multiple callbacks, the scheduler confirms the triggered callback and deletes others from the callback list. For example, if the image is correctly loaded, the $onerror$ callback will be deleted. Then, the scheduler will change the event status to ready so that the dispatcher can fetch and execute the event.

2) Event Cancellation: Once an event is scheduled by JSKERNEL in the event queue, a user-space script may request to cancel the event, e.g., via $clearTimeout$ and $cancelRequestAnimation$. When our scheduler receives a cancellation request, usually accompanied by the return value

in the event request stage, the scheduler will look up the event based on the ID field. Then, there are three possible cases. First, the event has not happened in the browser. If so, the scheduler will first cancel the event by calling the corresponding cancellation API and then marked the event as cancelled in the queue. Second, the event has happened in the browser and confirmed in the event queue, but not invoked by our dispatcher. If so, the scheduler will change the event status to cancelled directly. Lastly, the event has already been invoked by the dispatcher. If so, the scheduler will ignore the request.

3) Event Dispatching: The dispatcher is essentially an event loop that keeps fetching events from the event queue following their predicted time. If the status of the fetched event is ready, the dispatcher will invoke the event callback with the correct execution context and arguments, and then removes the event from the event queue. If the status of the fetched event is pending, the dispatcher will wait for the event to become ready. If the status of the fetched event is cancelled, the dispatcher will directly discard the event.

E. Thread Management

Different JavaScript threads, i.e., WebWorkers, have their own runtime, e.g., *self* object, and can freely communicate with the main thread. Therefore, JSKERNEL takes control of thread management and puts it into the kernel. We now first describe how to create kernel and user threads, and then introduce the communication between different threads.

1) Kernel and User Thread Creation: The thread manager of JSKERNEL provides a customized interface for the user-space script to create a thread. For example, when a user-space script constructs a new Web Worker, the the thread manager in the kernel code, will first construct a thread object to represent a kernel thread. The thread object contains four fields: status, ID, src, and kernel worker. The status field indicates whether the kernel thread has started ("started" status), loaded the user thread ("ready" status), or closed ("closed" status) due to either main or user thread request. The ID field represents a unique identifier for the kernel thread, the src field the user thread source, and the kernelWorker field a WebWorker responsible for the kernel thread. Next, the thread manager will create and then communicate with the kernel thread.

The kernel thread, once created, will prepare an environment for the user thread. Specifically, the kernel thread wraps all the timing related objects and APIs, available in a WebWorker, to an anonymous closure, which is just like what the kernel code does for the main thread. Note that a kernel thread maintains a separate event queue and clock from the main thread, i.e., the scheduling and clock ticking follow APIs and events in this kernel thread only (not the main thread or other kernel threads). Once the thread environment is prepared, the kernel thread will import the user-space thread with the source passed from the kernel code in main thread.

2) Thread Communication: Similar to OS's interprocess communication, there are two types of thread communication in browser: message passing, i.e., $postMessage$, and shared memory, i.e., $SharedArrayBuffer$. The former is very popular and widely used—and therefore JSKERNEL also adopts it for kernel communication; the latter is rarely used and currently disabled in many browsers due to Spectre attack. We discuss both communications.

First, JSKERNEL adopts message passing for both user-space and kernel-space communication. For example, JSKERNEL passes the user thread source from the main thread to one kernel thread via the kernel-space communication. Because there only exists one channel, i.e., the $postMessage$ and $onmessage$ one, between two threads, we create an overlay upon the channel. Specifically, JSKERNEL wraps the original object under a new object and uses a special field, i.e., a type field, in the object to indicate whether it is a kernel- or user-space communication. A kernel-space communication will be directly handled by the kernel code, and a user-space will be handled by the scheduler in each thread. Currently, JSKERNEL supports two types of kernel-space communication: exchanging a clock and passing thread source.

Second, browsers support $SharedArrayBuffer$ for two threads to share a chunk of memory, which may also be used for a fine-grained timer [12]. JavaScript cannot directly access contents of $SharedArrayBuffer$, but has to rely on yped array or $DataView$. Just like the Worker object in Listing 5, JSKERNEL provides a customized interface to access $SharedArrayBuffer$ contents so that every access is redirected to the kernel and put into the event queue.

IV. SECURITY ANALYSIS

We now evaluate the robustness of JSKERNEL along with depicting the corresponding security policies to handle web concurrency attacks. As a comparison, we evaluate five browsers—namely Google Chrome, Firefox, Microsoft Edge, Tor Browser [17], DeterFox [14] and Fuzzyfox [6]—and one browser extension, i.e., Chrome Zero [3]. It is worth noting that when we mention JSKERNEL, we refer to all three extensions on Firefox, Chrome and Edge. As the capability in defending against timing attacks for all three extensions are the same, for simplicity, we uniformly use the term JSKERNEL.

In the rest of the section, we start from implicit clocks in Section IV-A and cover web concurrency attacks in Section IV-B.

A. Implicit Clocks as a Web Concurrency Attack

In this section, we evaluate the robustness of JSKERNEL with a deterministic scheduling policy against timing attacks using different implicit clocks. A summary of the evaluation results is shown in Table I, i.e., JSKERNEL can defend against all existing attacks, while many of them can only defend against a limited number.

1) setTimeout as an implicit clock: In this part, we adopt setTimeout as an implicit clock and evaluate it against three types of timing attacks.

- *Cache attack.* A cache attack is where an adversary steals the contents being cached in the system—if certain contents have been flushed out from the cache, the access time to

Table I: Evaluation of Defenses against Web Concurrency Attacks

Attacks	Legacy Three†	Fuzzyfox	DeterFox	Tor Browser	Chrome Zero	JSKERNEL
setTimeout as the implicit clock						
Cache Attack [7]	✗	✗	✓	✗	✗	✓
Script Parsing [8]	✗	✗	✓	✗	✗	✓
Image Decoding [8]	✗	✗	✓	✗	✗	✓
Clock Edge [6]	✗	✓	✓	✗	✗	✓
requestAnimationFrame as the implicit clock						
History Sniffing [9]	✗	✗	✓	✗	✗	✓
SVG Filtering [9]	✗	✗	✓	✗	✗	✓
Floating Point [10]	✗	✓	✓	✓	✗	✓
Loopscan [11]	✗	✗	✓	✗	✗	✓
CSS Animation [12]	✗	✓	✓	✗	✗	✓
Video/WebVTT [6]	✗	✓	✓	✓	✗	✓
Other web concurrency attacks						
CVE-2018-5092	✗	✗	✗	✗	✗	✓
CVE-2017-7843	✗	✗	✗	✗	✗	✓
CVE-2015-7215	✗	✗	✗	✗	✗	✓
CVE-2014-3194	✗	✗	✗	✗	✓	✓
CVE-2014-1719	✗	✗	✗	✗	✓	✓
CVE-2014-1488	✗	✗	✗	✗	✗	✓
CVE-2014-1487	✗	✗	✗	✗	✗	✓
CVE-2013-6646	✗	✗	✗	✗	✓	✓
CVE-2013-5602	✗	✗	✗	✗	✗	✓
CVE-2013-1714	✗	✗	✗	✗	✗	✓
CVE-2011-1190	✗	✗	✗	✗	✓	✓
CVE-2010-4576	✗	✗	✗	✗	✓	✓

†: "Legacy Three" refers to three commercial, legacy browsers, i.e., Firefox, Chrome and Edge; ✓: The defense can prevent the attack; ✗: The defense is vulnerable.

Table II: Averaged, Measured Time of Different Targets under Varied Attacks (SVG filtering attack: averaged image loading time with different resolutions; Loopscan attack: maximum Measured Event Interval; note that all the numbers are averaged from multiple repeated experiments.)

Defense	**SVG Filtering**		**Loopscan Attack**	
	Low Resolution	High Resolution	google	youtube
Chrome	16.66 ms	18.85 ms	4.5 ms	8.8 ms
Firefox	16.27 ms	17.12 ms	50 ms	74 ms
Edge	23.85 ms	25.66 ms	20.8 ms	21.1 ms
Fuzzyfox	109.09 ms	145.45 ms	200 ms	500 ms
Tor Browser	16.63 ms	17.81 ms	500 ms	600 ms
Chrome Zero	15.71 ms	21.63 ms	12.8 ms	8.1 ms
JSKERNEL	10 ms	10 ms	1 ms	1 ms

such contents will be larger than the time to access unflushed ones. That said, the secret being targeted for extraction in a cache attack is the access time of specific contents. Web-level cache attacks were first proposed by Oren et al. [7], where an attacker accesses a specific shared storage, measures the latency, and guesses the website that the user has visited before. In this paper, we adopt a simplified version of the cache attack, i.e., measuring the access time of flushed and unflushed contents, using all the defenses. JSKERNEL can defend against the attack via a deterministic time.

- *DOM-based side channel attacks.* This timing attack proposed by van Goethem et al. [8] is to infer the size of cross-origin resources, such as the number of the user's social network Friends, by appending them to the Document Object Tree (DOM) tree and measuring the loading time. Hence, the secret, which the adversary tries to steal, is the loading time of DOM operations, such as appending children. There are different types of DOM elements that the adversary may want to steal—specifically, van Goethem et al. propose two types of attacks, i.e., script parsing and image decoding. The former loads a cross-origin resource as a script and the latter as an image. Both attacks, as shown in Table I, are still possible in all the existing defenses except for JSKERNEL and DeterFox, which adopt determinism to defeat timing attacks.

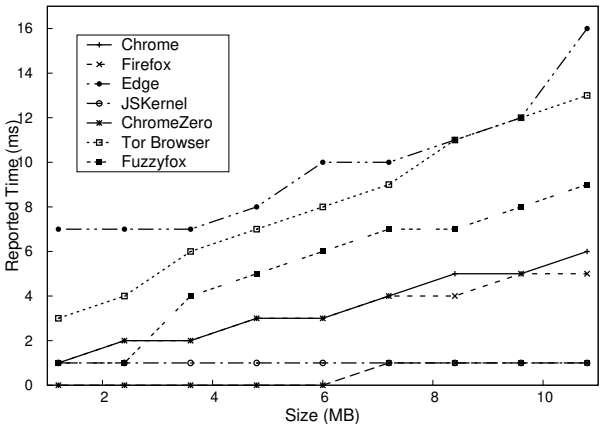

Figure 2: Script Parsing Attack with Asynchronous Clock (Except for JSKERNEL, the reported parsing time measured by the callback of setTimeout increases for all other defenses when the size of the file increases.)

The evaluation results in Figure 2 show that existing defenses other than JSKERNEL are vulnerable to script parsing attacks. Specifically, when the file size increases, the loading time also does so. That is, an adversary can infer the file size based on the loading time. Firefox, Chrome and Edge show a linear line, i.e., it is easy to differentiate fine-grained file size in both browsers. Tor Browser, Fuzzyfox and Chrome Zero raise the bar, making it harder to differentiate files with small size difference, but it is still possible to infer two files with larger than 1MB difference.

2) Animation-related timing attack using requestAnimationFrame or CSS Animation as an implicit clock: An animation-related timing attack, such as history sniffing [9], SVG filtering [14], floating point [10], relies on the $requestAnimationFrame$ API or CSS Animation [12] to measure the time to launch a repainting related operation, i.e., a secret. The time may be used to further infer cross-origin contents, such as pixels in a cross-origin image. For example, the history sniffing attack can be used to differentiate the color of a visited and unvisited link, thus determining whether the link is visited by the browser user. The SVG filtering attack mentioned in the DeterFox paper [14] can be used to differentiate two images with drastically different resolutions. The floating-point attack [10] is designed to steal pixels from a cross-origin image. The original attacks have been fixed by

Chrome and Firefox, and the version that we used here is a combination of the original attack and pixel stealing one [9].

We take a closer look at the SVG filtering attack in Table II (SVG Filtering column). We run each test for 25 times against each defense and take the average duration as the measurement results. The evaluation results show that this side channel still exists in all other defenses except for JSKERNEL. Specifically, the adversary can easily differentiate two images in Chrome, Firefox, Edge, Chrome Zero, and Tor browser with a few runs. Fuzzyfox does increase the bar, because it adds much noise to the execution time; however, an adversary can still average the results of 25 runs and differentiate two images with different resolutions.

3) Loopscan Attack: Loopscan [11] is a novel attack that monitors the event loop usage pattern to infer the domain name of cross-origin websites visited by the user. For example, the usage pattern of google.com is different from the one of youtube.com, and therefore an adversary can infer what websites are visited by the victim user. We adopt the original implementation of Loopscan to evaluate existing defenses. For simplicity, we only record the maximum event interval and evaluate existing defenses' capability to differentiate two websites. Table II (Loopscan Attack column) shows the evaluation results: except for JSKERNEL, all other defenses are vulnerable to the Loopscan attack. That is, the maximum event intervals of these defenses are different for youtube.com and google.com, and thus an adversary can infer the website name based on the interval.

4) Clock edge attack: Clock edge attack [6] measures the duration of a cheap operation, such as $i++$, by using a coarse-grained clock. Specifically, an adversary can count the number of a cheap operation between two edges of one tick in the coarse-grained clock—and the calculate the weight of the cheap operation by dividing the number by the tick value of the coarse-grained clock. Then, the cheap operation can serve as a fine-grained clock to measure a secret.

The evaluation results in Table I show that JSKERNEL can defend against the clock edge attack. The reason is that the time interval between two coarse-grained clock API calls is determined by the number of API calls but not the number of the cheap operations. By contrast, the clock edge attack provides a more accurate timer in Chrome, Firefox and Tor Browser. Fuzzyfox does defend against the clock edge attack as claimed in the paper.

B. Other Web Concurrency Attacks

We herewith evaluate the capability of JSKERNEL to defend against other web concurrency attacks. The methodology of finding and evaluating web concurrency attacks is as follow. First, we search the keyword "worker" and a browser name, such as Firefox and Chrome, on the National Vulnerability Database (NVD) and then manually go through all the vulnerabilities to confirm their relationship to web concurrency attack. Note that this may not be a complete list of all the web concurrency attacks but it is the best we can do to find web concurrency attacks. Second, we download the vulnerable version of the browser together, find the available attacks online, e.g., on Bugzilla, and then evaluate the corresponding defenses. Since some older browsers do not support new features, we replace these new features with old ones correspondingly. Note that some vulnerabilities are platform specific—for example, CVE-2018-5092 can only be triggered on Windows 10.

A high-level overview of the results is shown in Table I. The native defenses are not robust as none of them are equipped to consider web concurrency attacks except for simpler cases of timing with implicit clocks. Chrome Zero can defend against some vulnerabilities at the price of reduced functionalities as Chrome Zero only adopts a polyfill implementation of a web worker. We now present some examples to illustrate why JSKERNEL can defend against these vulnerabilities.

- CVE-2013-1714 [18] is a vulnerability that violates same-origin policy, i.e., a worker thread can send an XMLHttpRequest to web servers with any origin. The condition to trigger the vulnerability is that the request needs to come from a worker thread. Therefore, JSKERNEL enforces a policy to check the origins for all the requests coming from a web worker.
- CVE-2013-5602 [19] is a vulnerability that refers to a null pointer when an adversary assigns an $onmessage$ callback function to a Web Worker object. JSKERNEL enforces a policy to avoid assigning an $onmessage$ callback by hooking both the $setter$ function of $onmessage$ and $setEventListener$.
- CVE-2014-1488 [20] is a use-after-free vulnerability, in which the worker thread passes a transferable ArrayBuffer to the main thread but will free the ArrayBuffer once it is terminated. The condition to trigger the vulnerability is that the worker thread needs to first pass a transferable to the main thread and then be terminated. Therefore, the policy enforced by JSKERNEL is that if the worker thread passes a transferable object, the worker will only be terminated at the user level, but the kernel level will still maintain the worker to avoid the triggering condition.
- CVE-2014-1487 [21] and CVE-2015-7215 [22] are two similar information disclosure vulnerabilities that show cross-origin information in the error message of a worker thread creation and the $importScripts()$ function of a worker thread. JSKERNEL enforces a policy that sanitizes the error message of $onerror$ callback function and $importScripts()$ by throwing a new message without the cross-origin information and ensuring security.
- CVE-2017-7843 [23] is a vulnerability in which the access to indexedDB in private browsing mode is not deleted after existing. Therefore, the policy enforced by JSKERNEL is to avoid access to indexedDB during private browsing mode to obey the mode's specification.

V. SYSTEM EVALUATION

In this section, we evaluate JSKERNEL based on two metrics: performance overhead and compatibility.

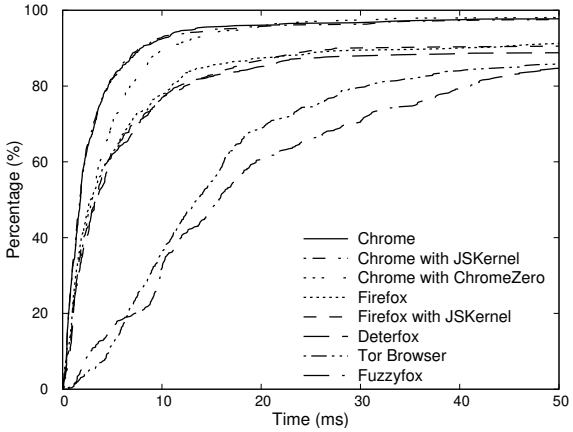

Figure 3: Cumulative Distribution Function (CDF) of Loading Time of Top 500 Alexa Websites (Browsers with JSKERNEL and DeterFox incur the least overhead; Tor Browser and Fuzzyfox are the slowest; Chrome Zero incurs more overhead than JSKERNEL.)

A. Performance Overhead

We evaluate the performance overhead using both micro- and macro-benchmarks. The micro-benchmark evaluates the performance overhead of specific APIs, especially those related to web concurrency attacks; the macro-benchmark evaluates top Alexa websites using both loading times and checkpoints specified in the Raptor test. Note that all the experiments are performed on a Linux machine with an Intel(R) 2.30GHz Core(TM) i5-6300HQ CPU, 8 GB memory, and an ADSL network with 9.5 Mbit/s bandwidth.

1) Micro-benchmark: Dromaeo and Worker Test: We first evaluate JSKERNEL extensions with Google Chrome using Dromaeo [15], a comprehensive JavaScript performance benchmark with many micro-level test cases, such as mathematical calculations, data structure manipulations, and DOM operations. Overall, the performance drops 1.99% on average after installing JSKERNEL on Google Chrome and the median performance reduction is 0.30%. The DOM Attribute Test incurs the largest overhead, i.e., 21.15% decrease in the performance, because this test needs to traverse through the kernel and the website JavaScript for many times, which brings overhead.

As Dromaeo does not have any web worker involved, we also tested JSKERNEL extension with Google Chrome under a Worker benchmark (http://pmav.eu/stuff/javascript-webworkers/). Specifically, we created 16 workers and measured the time to create these workers with 5 repeat experiments—the average overhead is 0.9% with and without JSKERNEL extension.

2) Macro-benchmark: Alexa Top 500 Websites: We evaluated the performance of JSKERNEL using Alexa Top 500 websites. Specifically, we used Selenium [24], a browser automation tool, to visit Alexa websites and record the timestamp before the visit and the timestamp when *onload* event is fired. We then calculated the interval, i.e., the loading time of the website, on our experiment machine. Each experiment is performed three times to obtain an average result. Figure 3 shows the cumulative distribution function (CDF) of the loading time of Alexa websites with seven different browsers, i.e., Chrome, Chrome with JSKERNEL, Chrome with Chrome Zero, Firefox, Firefox with JSKERNEL, DeterFox [14], Tor Browser [17], and Fuzzyfox [6].

Table III: Average Website Loading Time in Raptor-tp6-1 (C: Chrome and F: Firefox). All numbers in the table are in milliseconds (ms).

Subtest	Chrome (C)	JSKERNEL (C)	Firefox (F)	JSKERNEL (F)
Amazon	107.2±12.8	109.3±11.4	809.1±50.6	831.9±57.9
Facebook	178.8±9.5	172.1±15.7	1,018.9±109.1	1,005±238.3
Google	48.3±3.2	51.3±3.5	400.7±107.9	425.4±105.4
Youtube	298.9±110.6	308.9±100.3	1,249.8±158.4	1,136.8±135.1

There are four things worth noting. First, JSKERNEL adds minimal, non-observable overhead to the web browsers. Specifically, the curve of JSKERNEL with Chrome and Firefox is very close to the original browser with no observable, statistically-significant overhead. Second, although the performance of DeterFox is also similar to Firefox, DeterFox only works as a Firefox variance. That is, it remains unclear how to integrate DeterFox with Google Chrome, which may require significant engineering work. Third, both Tor Browser and Fuzzyfox, which are based on and modified from Firefox, incur non-negligible overhead, because they add noises to the browser. In addition, similar to DeterFox, it is unclear how to integrate these two approaches with Google Chrome either. Lastly, both JSKERNEL and Chrome Zero can be deployed as a Chrome extension, but JSKERNEL incurs much less overhead compared with Chrome Zero.

3) Macro-benchmark: Raptor Loading Tests: In addition to the aforementioned experiment on Alexa Top 500 Websites, we also run Raptor loading tests [25], i.e., raptor-tp6-(1–7), to evaluate the performance of JSKERNEL. The reason that we use this test is that some modern websites continue loading after the *onload* event via JavaScript—Raptor loading tests, which adopt a hero element to indicate the loading time, can capture such loading tasks performed by websites. Specifically, we load each subtest 25 times and skip the first result due to the involvement of opening a tab. The average loading overhead for JSKERNEL on Chrome (as indicated by the loading of the hero element) is 2.75% and on Firefox 3.85%.

We also listed the detailed numbers for raptor-tp6-1 in the Table III. There are two things worth noting. First, the loading time with JSKERNEL could be smaller than the one without JSKERNEL, such as in the case of Facebook and Youtube (Firefox), because elements loaded in JSKERNEL may follow a specific sequence in which the hero element is loaded early. Second, the time differences with and without JSKERNEL are smaller than the standard deviation, i.e., the overhead is small enough.

B. Compatibility

In this subsection, we evaluate the compatibility of JSKERNEL with legacy websites and JavaScript applications.

1) API Specific Test: In order to find legacy JavaScript applications with certain APIs, we rely on CodePen [26], a social development environment for front-end designers and developers. Specifically, CodePen provides a search interface, in which we can type an API name, such as $performance.now$, and then CodePen will return a list of applications using that searched API. In this experiment, we obtain the top five applications returned by CodePen as our test dataset when searching a corresponding API in CodePen.

Our evaluation methodology works as follows. We ask a student to first run the application in four different browsers: Firefox, Fuzzyfox, DeterFox, and a Firefox with JSKERNEL installed. The student needs to interact with the application and play with its interface. Then, the student will tell us the experience when running the application in four different browsers.

We now describe a summary of the API specific test. First, when we compare JSKERNEL with the other two defenses, i.e., Fuzzyfox and DeterFox, JSKERNEL is the one with the least observable differences. Specifically, Fuzzyfox executes 13 apps out of 20 apps with observable differences, DeterFox 7 out of 20, and JSKERNEL 4 out of 20. All the differences in JSKERNEL are either a higher or lower FPS caused by the usage of the synchronous timer $performance.now$, because $performance.now$ is mainly used for fine-grained time-related operations, such as timing and generating an animation with a time constraint.

Second, both Fuzzyfox and DeterFox introduce non-time related differences, such as loading errors of the app, images, objects, and background. By contrast, JSKERNEL introduces only time-related differences, such as higher frames per second (FPS) and faster clock. The reason is that both Fuzzyfox and DeterFox operate on the browser source code, which is written in C or C++. That is, a small engineering error may cause the browser to crash. JSKERNEL is written in JavaScript and therefore has a good memory protection provided by the browser.

2) Semi-automated Compatibility Test on Alexa Top 100 Websites: In this section, we test the compatibility of JSKERNEL with Alexa Top 100 websites. Here is our methodology. We visit each website twice, one with JSKERNEL and the other without JSKERNEL, on Google Chrome Browser. During each visit, we output the document object tree (DOM) of the website and serialize the structure into a string. Then, we compare these two strings using cosine similarity: if the similarity is larger than 99%, we will consider that these two visits render the same results; if not, we will ask a human to look at both rendering results.

Our evaluation results show that 90% of websites have larger than 99% similarity scores if visited with and without JSKERNEL. We manually checked the rest ten websites, which are all caused by dynamic contents, such as ads. At the same time, we visit these ten websites twice directly on Google Chrome without JSKERNEL and calculate the similarity score. The score is very close, i.e., less than 2% difference, to the one obtained from JSKERNEL compatibility test.

3) A Week-long User Experience Test: In this section, we present a week-long user experience test on compatibility. Specifically, we ask a student that is not on the author list to install our JSKERNEL on a Chrome browser on his laptop and browse the Internet for a week. In the first two days, the student does experience three issues, one on Overleaf, an online Latex editor, one on Google Calendar, and another on Google Map—all the issues are fixed in the current version. The first issue is that the student cannot compile a PDF file on overleaf. We looked into it and found that the reason is that our web worker implementation has a bug in dealing with an absolute path. The second issue is that all the Mondays on Google Calendar are shown as Wednesdays due to a bug in our Date object implementation. The last issue is that one Google Map Worker accesses the Worker location, which falsely points to our kernel worker due to a bug. After we fixed all the bugs, the student did not experience any other compatibility issues in the rest five days of using the extension.

VI. DISCUSSION

In this section, we discuss several issues related to JSKERNEL. First, we consider the robustness of JSKERNEL against self-modifying code, i.e., when the adversary knows that the client browser installed JSKERNEL to prevent attacks. We believe that even if the adversary knows that JSKERNEL is present, the adversary cannot bypass the protection enforced by it. The reasons are fourfold. (i) All the JSKERNEL code and attack-related APIs are encapsulated inside the JSKERNEL kernel so that an adversary cannot access them. (ii) Even if the adversary modifies the interface of JSKERNEL provided by the kernel, such modification will only affect the website's functionality—but still the adversary cannot access corresponding APIs. These are encapsulated inside JSKERNEL kernel. (iii) JSKERNEL injects JSKERNEL kernel into every new JavaScript context, such as a newly-opened window and an iframe. (iv) JSKERNEL obtains all the JavaScript functions and redefines them using a customized pointer. JSKERNEL also adopts $Object.freeze()$ to avoid any pollutions to the prototype property of system objects (e.g., Array and Object). In the future, we plan to follow Bhargavan et al. [27] to write JSKERNEL in a defensive JavaScript subset.

Second, we discuss the capability of JSKERNEL in defending against unknown vulnerabilities. JSKERNEL can defend against unknown timing attacks because the scheduler arranges all asynchronous events in a deterministic order. At present, JSKERNEL only defends against other web concurrency attacks on a case-by-case base, because JSKERNEL requires vulnerability-specific policies. We leave it as a future work to automatically extract policies for a new vulnerability.

Third, we discuss the difference between the triggering condition and the the underlying vulnerability. Web concurrency attacks capture the nature of vulnerability triggering condition, i.e., the needs of concurrency information from different threads to trigger a vulnerability, while the underlying vulnerability may differ, which could be a user-after-free, a cross-site information leak or a privilege escalation.

VII. Related Works

We first present existing defenses against timing attacks and low-level attacks. Next, we overview existing third-party JavaScript isolation works.

1) Defense against Timing Attacks: There are three categories of defenses, i.e., attack surface reduction, fuzzy time, and determinism, to prevent timing attacks. First, Snyder et al. [1] show that one can disable certain JavaScript APIs, such as WebGL, Audio, and WebAssembly, to reduce the attack surface. As one of its applications, Snyder et al. can also be used to prevent timing attacks by disabling timing related APIs. Though being effective, such approach will bring compatibility issues for these websites adopting disabled APIs.

Second, Kohlbrenner et al. [6] randomize the performance of executions by introducing pause tasks into the browser's event queue. Their prototype browser, Fuzzyfox, obfuscates the duration for a specific execution. Inherited from the idea of Kohlbrenner et al. [6], JavaScript Zero [3] proposes to redefine certain timing-related APIs and introduce fuzzy time in the browser extension level. Such approach significantly increases the protection range because any users can deploy the proposed extension on a daily basis. However, because JavaScript Zero still adopts fuzzy time, it cannot fundamentally prevent timing attacks just as Kohlbrenner et al.

Lastly, deterministic execution model is another strategy to mitigate timing attacks. The deterministic browser project [14] is the first attempt to apply deterministic execution model to the modern browser. As a comparison, JSKernel, can be installed on any existing web browsers, such as Google Chrome, Firefox and Microsoft Edge, to protect users, and also able to defend against low-level attacks. At the same time, researchers also propose deterministic models for specific timing channels, such as floating-point operations. For example, CTFP [28] uniformizes the execution of certain heavy-weight floating-point operations. As a comparison, JSKernel can prevent all types of timing channels including floating-point ones.

It is worth noting that determinism [29], [30], [31], [32], [33], [34], [35] was proposed long before DeterFox [14] to prevent timing attacks in general. Apart from defending against timing attacks, the determinism technology is also used to schedule multi-thread programs to increase their stability [36], [37], [38], [39], [40], [41]. Additionally, deterministic virtual clock is used to guard the sequence of execution in distributed systems [42].

2) Defense against Low-level Attacks: Snyder et al. [1] disable certain JavaScript APIs to reduce attack surface but also at the price of reduced functionalities. BrowserShield [2] proposes to defend against low-level, zero-day browser vulnerability via rewriting JavaScript code. Although the rewriting is effective in defending against many low-level vulnerabilities, it at the same time incurs significant overhead because it instruments every JavaScript operations. JShield [43] modifies a browser to enforce their signatures for security—the defense is effective but also brings compatibility issues. That is, JShield is only applicable to one type of browser and needs to be updated with every browser version. As a comparison, JSKernel is compatible with three browsers and does not need any updates for a new browser version.

3) Third-party JavaScript Isolation: Third-party JavaScript isolation, such as AdJail [44], AdSentry [45], JSand [46], PAD [47], AdJust [48], and Virtual Browser [49], provides a sandbox for third-party JavaScript like ads. These works can successfully prevent third-party JavaScript from tampering with trusted contents, such as first-party JavaScript.

Currently, JSKernel adopts an anonymous closure for isolation, but can adopt any of the aforementioned isolation techniques proposed by prior work. The contribution of JSKernel is the capability of enforcing a customized event scheduling policy to defend against web concurrency attacks—the isolation component of JSKernel is an orthogonal problem from event scheduling.

VIII. Conclusion

In this paper, we proposed JSKernel, the first approach to introduce a kernel concept that enforces the order of JavaScript execution of threads and events to defend against web concurrency attacks. We implemented a prototype system as extensions to three major commercial web browsers, i.e., Firefox, Google Chrome, and Edge and made it open-source. Our evaluation shows that JSKernel is fast, robust to a variety of attacks and is backward compatible with existing web applications.

Acknowledgement

We want to thank our shepherd, Neeraj Suri, and anonymous reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) grant CNS-18-54001. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

References

[1] P. Snyder, C. Taylor, and C. Kanich, "Most websites don't need to vibrate: A cost-benefit approach to improving browser security," in *Proceedings of the 2017 ACM CCS*, 2017.

[2] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "Browsershield: vulnerability-driven filtering of dynamic html," in *OSDI: USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[3] M. Schwarz, M. Lipp, and D. Gruss, "Javascript zero: Real javascript and zero side-channel attacks," in *NDSS*, 2018.

[4] Canvas defender. https://addons.mozilla.org/en-US/firefox/addon/no-canvas-fingerprinting/.

[5] Disable webrtc. https://addons.mozilla.org/en-US/firefox/addon/happy-bonobo-disable-webrtc/.

[6] D. Kohlbrenner and H. Shacham, "Trusted browsers for uncertain times," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 463–480.

[7] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox - practical cache attacks in javascript," *CoRR*, vol. abs/1502.07373, 2015. [Online]. Available: http://arxiv.org/abs/1502.07373

[8] T. Van Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," in *Proceedings of the 22nd ACM CCS*, 2015.

[9] P. Stone., "Pixel perfect timing attacks with html5," Tech. Rep., 2013. [Online]. Available: https://www.contextis.com/resources/white-papers/pixel-perfect-timing-attacks-with-html5

[10] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

[11] P. Vila and B. Kopf, "Loophole: Timing attacks on shared event loops in chrome," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[12] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript," in *Financial Cryptography and Data Security (FC)*, 2017.

[13] Web workers - use after free in nswrapper-cache::getwrapperpreservecolor(). https://bugzilla.mozilla.org/show_bug.cgi?id=1418074.

[14] Y. Cao, Z. Chen, S. Li, and S. Wu, "Deterministic browser," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. ACM, 2017, pp. 163–178.

[15] J. Resig, "Dromaeo javascript performance test suite," Tech. Rep. [Online]. Available: http://dromaeo.com/

[16] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan, "Concurrency attacks," in *Presented as part of the 4th {USENIX} Workshop on Hot Topics in Parallelism*, 2012.

[17] (2018) Tor browser. https://www.torproject.org/projects/torbrowser.html.en.

[18] Cross domain policy override using webworkers. https://bugzilla.mozilla.org/show_bug.cgi?id=879787.

[19] Asan segv on unknown address in worker::seteventlistener. https://bugzilla.mozilla.org/show_bug.cgi?id=897678.

[20] Firefox reproducibly crashes when using asm.js code in workers and transferable objects. https://bugzilla.mozilla.org/show_bug.cgi?id=950604.

[21] Cross-origin information disclosure with error message of web workers. https://bugzilla.mozilla.org/show_bug.cgi?id=947592.

[22] Cross-origin information disclosure with error message of web workers importscripts(). https://bugzilla.mozilla.org/show_bug.cgi?id=1160890.

[23] fingerprinting users in private window using web-worker + indexeddb. https://bugzilla.mozilla.org/show_bug.cgi?id=1410106.

[24] (2018) Selenium. https://www.seleniumhq.org/.

[25] Performance sheriffing/raptor. https://wiki.mozilla.org/Performance_sheriffing/Raptor.

[26] (2018) Codepen - front end developer playground and code editor in the browser. https://codepen.io/.

[27] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Defensive javascript," in *Foundations of Security Analysis and Design VII*. Springer, 2014, pp. 88–123.

[28] M. Andrysco, A. Nötzli, F. Brown, R. Jhala, and D. Stefan, "Towards verified, constant-time floating point operations," in *Proceedings of the 2018 ACM CCS*, 2018.

[29] A. Aviram, S. Hu, B. Ford, and R. Gummadi, "Determinating timing channels in compute clouds," in *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, ser. CCSW '10, 2010.

[30] W. Wu, E. Zhai, D. Jackowitz, D. I. Wolinsky, L. Gu, and B. Ford, "Warding off timing attacks in deterland," *CoRR*, vol. abs/1504.07070, 2015. [Online]. Available: http://arxiv.org/abs/1504.07070

[31] M. Huisman, P. Worah, and K. Sunesen, "A temporal logic characterisation of observational determinism," *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pp. 13 pp.–3, 2006.

[37] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang, "Efficient deterministic multithreading through schedule relaxation," in *SOSP*, 2011.

[32] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *Proceedings of the 13th IEEE Workshop on Computer Security Foundations*, ser. CSFW '00, 2000.

[33] G. Smith and D. Volpano, "Secure information flow in a multi-threaded imperative language," in *POPL*, 1998.

[34] S. Zdancewic and A. C. Myers, "Observational determinism for concurrent program security," in *CSFW*, 2003.

[35] D. Volpano and G. Smith, "Eliminating covert flows with minimum typings," in *Computer Security Foundations Workshop, 1997. Proceedings., 10th*. IEEE, 1997, pp. 156–168.

[36] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant, "Parrot: a practical runtime for deterministic, stable, and reliable threads," in *SOSP*, 2013.

[38] H. Cui, J. Wu, C.-C. Tsai, and J. Yang, "Stable deterministic multithreading through schedule memoization," in *Proceedings of the 9th USENIX OSDI*, 2010.

[39] T. Liu, C. Curtsinger, and E. D. Berger, "Dthreads: efficient deterministic multithreading," in *SOSP*, 2011.

[40] M. Olszewski, J. Ansel, and S. P. Amarasinghe, "Kendo: efficient deterministic multithreading in software," in *ASPLOS*, 2009.

[41] J. Yang, H. Cui, J. Wu, Y. Tang, and G. Hu, "Making parallel programs reliable with stable multithreading," *Commun. ACM*, vol. 57, no. 3, pp. 58–69, Mar. 2014.

[42] D. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 404–425, 1983.

[43] Y. Cao, X. Pan, Y. Chen, and J. Zhuge, "JShield: Towards real-time and vulnerability-based detection of polluted drive-by download attacks," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC, 2014.

[44] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan, "Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements," in *Proceedings of the 19th USENIX Conference on Security*, ser. USENIX Security'10, 2010.

[45] X. Dong, M. Tran, Z. Liang, and X. Jiang, "Adsentry: comprehensive and flexible confinement of javascript-based advertisements," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 297–306.

[46] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, "Jsand: complete client-side sandboxing of third-party javascript without browser modifications," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 1–10.

[47] W. Wang, Y. Kwon, Y. Zheng, Y. Aafer, I.-L. Kim, W.-C. Lee, Y. Liu, W. Meng, X. Zhang, and P. Eugster, "Pad: Programming third-party web advertisement censorship," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 240–251. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155596

[48] W. Wang, I. L. Kim, and Y. Zheng, "Adjust: Runtime mitigation of resource abusing third-party online ads," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, pp. 1005–1015.

[49] Y. Cao, Z. Li, V. Rastogi, Y. Chen, and X. Wen, "Virtual browser: A virtualized browser to sandbox third-party javascripts with enhanced security," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '12, 2012.