Spook.js: Attacking Chrome Strict Site Isolation via Speculative Execution

Ayush Agarwal*
University of Michigan
ayushagr@umich.edu

Sioli O'Connell*
University of Adelaide
sioli.oconnell@adelaide.edu.au

Jason Kim
University of Michigan
nosajmik@umich.edu

Shaked Yehezkel
Tel Aviv University
shakedy@mail.tau.ac.il

Daniel Genkin
University of Michigan
genkin@umich.edu

Eyal Ronen
Tel Aviv University
eyal.ronen@cs.tau.ac.il

Yuval Yarom
University of Adelaide and Data61
yval@cs.adelaide.edu.au

***Abstract*—The discovery of the Spectre attack in 2018 has sent shockwaves through the computer industry, affecting processor vendors, OS providers, programming language developers, and more. Because web browsers execute untrusted code while potentially accessing sensitive information, they were considered prime targets for attacks and underwent significant changes to protect users from speculative execution attacks. In particular, the Google Chrome browser adopted the *strict site isolation* policy that prevents leakage by ensuring that content from different domains are not shared in the same address space.**

The percieved level of risk that Spectre poses to web browsers stands in stark contrast with the paucity of published demonstration of the attack. Before mid-March 2021, there was no public proof-of-concept demonstrating leakage of information that is otherwise inaccessible to an attacker. Moreover, Google's leaky.page, the only current proof-of-concept that can read such information is severely restricted to only a subsset of the address space and does not perform cross-website accesses.

In this paper we demonstrate that the absence of published attacks does not indicate that the risk is mitigated. We present Spook.js, a JavaScript-based Spectre attack that can read from the entire address space of the attacking web page. We further investigate the implementation of strict site isolation in Chrome, and demonstrate limitations that allow Spook.js to read sensitive information from *other* web pages. We further show that Spectre adversely affect the security model of extensions in Chrome, demonstrating usernames and passwords leaks from the LastPass password manager. Finally, we show that the problem also affects other Chromium based browsers, such as Microsoft Edge and Brave.

I. Introduction

Recent computer trends have significantly changed the way we use and distribute software. Rather than downloading installation packages, users now prefer to "live in their browser", where software is seamlessly downloaded, compiled, optimized, and executed by merely accessing a URL. Starting from humble origins, the browser is no longer a simple GUI for rendering text documents, but instead is more akin to a "mini operating system", complete with its own execution engines, compilers, memory allocators, and API calls to underlying hardware features. Perhaps most importantly, the browser has evolved into a highly-trusted component in almost any user-facing computer system, holding more secret data than any other computer program, except the operating system.

* Equal contribution, joint first authorship

Concurrent with the rise in importance of the browser, the rapid growth in complexity of computer systems over the past decades has resulted in numerous hardware security vulnerabilities [2, 6, 15, 21, 22, 24, 25, 26, 29, 45, 46, 47, 52, 61, 62, 63, 66, 67, 69]. Here, the attacker artificially induces contention on various system resources, aiming to cause faults or recover information across security boundaries. Perhaps the most known such incident was the discovery of Spectre [30] and Meltdown [35], which Google dubbed as a watershed moment in computer security [39].

Recognizing the danger posed by browser-based transient-execution attacks, Google has attempted to harden Chrome against Spectre. Introducing the concept of strict site isolation [51], Google's main idea is to isolate websites based on their domains, rendering mutually-distrusting pages in different memory address spaces. Aiming to further hinder attacks, Google elected to keep its JavaScript code in 32-bit mode, effectively partitioning the renderer's address space into multiple disjoint 4 GB heaps. It is hoped that even if a memory disclosure vulnerability is exploited, the use of 32-bit pointers will confine the damage to a single heap [19]. However, given the empirical nature of these countermeasures, in this paper we ask the following questions:

Is Chrome's strict site isolation implementation sufficient to mitigate browser-based transient execution attacks? In particular, how can attacker mount a transient execution attack that recovers sensitive information, despite Google's strict site isolation and 32-bit sandboxing countermeasures?

A. Our Contribution

In this paper we present Spook.js, a new transient-execution attack capable of extracting sensitive information despite Chrome's strict site isolation architecture. Moreover, Spook.js can overcome Chrome's 32-bit sanboxing countermeasures, reading the entire address space of the rendering process. Additionally, we demonstrate a gap between Chrome's eTLD+1-based consolidation policy and the same-origin policy typically used for security in web context. The discrepancy can results in mutually distrusting security domain co-residing in the same address space, allowing one subdomain to attack another. Going beyond Chrome, we show that Chromium-based browsers such as Edge and Brave are also vulnerable to Spook.js. We further show that Firefox's strict site isolation follows a similar eTLD+1 consolidation policy, but we leave the task of porting Spook.js to Firefox to future work.

Escaping 32-Bit Boundaries via Speculative Type Confusion. Although it executes in 64-bit mode, Chrome uses 32-bit addressing for its JavaScript architecture. This limits the information available to most Spectre-based techniques, as even a (speculative) out of bounds array index cannot escape the 4 GB heap boundary. To overcome this issue, Spook.js uses a type-confusion attack that allows it to target the entire address space. At a high level, our attack confuses the execution engine to speculatively execute code intended for array access on a carefully crafted malicious object. The malicious object is designed to place attacker-controlled fields where the code expects a 64-bit pointer, allowing aspeculative access to arbitrary addresses. We are not aware of any publication of speculative type-confusion attacks made so far.

Avoiding Deoptimization Events via Speculative Hiding. Even if a type-confusion attack is successful, the type of the malicious object does not match the expected array type. In response to such a mismatch, Chrome deoptimizes the array access code which Spook.js exploits, preventing subsequent applications of the attack. We overcome this issue by performing the entire type-confusion attack under speculation, running the attack inside a mispredicted `if` statement. Consequently, Chrome is completely oblivious to the type-confusion attack and its type mismatch, and does not deoptimize the code used for array access. This allows us to run our attack across multiple iterations, reading hundreds of kilobytes of data form the address space of the rendering process.

Applicability to Multiple Architectures. With the basic blocks of our attack in place, we proceed to show the feasibility of Spook.js across multiple architectures, including CPUs made by Intel, AMD and Apple. For Intel and Apple, we find that Spook.js can leak data at rates of around 500 Bytes per second, with around 96% accuracy. For AMD, we obtain similar leak rates assuming a perfect L3 eviction primitive for AMD's non-inclusive cache hierarchy, the construction of which we leave to future work.

Security Implications of eTLD+1 Based Consolidation. Having established the feasibility of reading arbitrary addresses from the address of Chrome's rendering processes, we now turn our attention to Chrome's eTLD+1 address space consolidation policy. Rather than using the same-origin policy, which considers two resources to be mutually-trusting if their entire domain name matches, Chrome uses a more relaxed policy that consolidates address spaces based on their effective top-level plus one (eTLD+1) domains.

We show that this difference is significant, demonstrating how a malicious web page (e.g., a user's homepage) located on some domain can recover information from login-protected domain pages displayed in adjacent tabs. Here, we show that a personal page uploaded to the University of Michigan can recover login-protected information from the University HR portal displayed in adjacent tabs, including contact information, bank account numbers, and even paycheck data.

Recovering Login Credentials and Cookies. Going beyond displayed information, we show how Spook.js can recover login credentials, both from Chrome's built-in password manager and from LastPass, a popular third-party extension. We also extend Spook.js to recover the domain's `HttpOnly` session cookies, allowing any malicious page in the domain to effectively siphon-off credentials as soon as it is opened by the target user. Here, we empirically contradict an explicit security goal of Chrome's strict site isolation, as the Google's strict site isolation paper explicitly states that `HttpOnly` cookies are not delivered to renderer processes [51, Section 5.1].

Exploiting Unintended Uploads. Tackling the case where a malicious presence on a domain is not possible, we show that user-uploaded cloud-content is often automatically transferred between different domains of the same provider. Here, we show how content uploaded to a `google.com` domain is actually stored by Google on `googleusercontent.com`, where it can be consolidated with personal webpages created on Google Sites. Empirically demonstrating this attack, we show the recovery of an image uploaded to a `google.com` domain, through a malicious Google Sites webpage.

Exploiting Malicious Extensions. Moving away from website consolidation, we port Spook.js into a malicious Chrome extension which requires no permissions. We show that Chrome fails to properly isolate extensions, allowing one extension to speculatively read memory of other extensions. We empirically demonstrate this on the LastPass extension, recovering both website-specific credentials as well as the vault's master password (effectively breaching the entire account).

Summary of Contributions. In this paper we make the following contributions:

- We weaponize the speculative execution attacks on the Chrome browser, demonstrating Spook.js, an attack that can read from arbitrary addresses within the rendering process's address space (Section III).
- We explore the limitations of Chrome's strict site isolation and demonstrate that consolidating websites into the same address space is risky, even when only performed in very restricted scenarios (Section IV).
- We study the implications of Spook.js on the security model of extensions in Chrome. We demonstrate that an unprivileged attack can recover the list of usernames and used passwords from a leading password manager (Section V).
- We show that Chromium-based browsers, such as Microsoft Edge and Brave are also vulnerable (Section VI).

B. Responsible Disclosure and Ethics

Disclosure. Shortly after submission, we will share a copy of the submission with the security teams of Intel, AMD, Chrome, Mozilla and LastPass. Experiments performed on the

University of Michigan systems were coordinated with the university's IT department and Chief Security Officer.

Ethics. Some experiments we conduct, require placing attack code on publicly-accessible web pages. To limit access to such pages and prevent capability leak and potential 0-days in the wild, we ensure that no links to attack pages are placed in any web page and that attack pages are only activated if the browser presents a specific cookie that we manually place in it. Data collection and inspection is done on the local machine and is never uploaded to external servers.

II. Background

A. Caches

To bridge the gap between the fast execution core and the slower memory, processors store recently accessed memory in fast caches. Most modern caches are set associative, meaning that the cache is divided into a number of sets, each of which is further divided into a fixed number of ways. Each way can store a fixed-size block of data, also called a cache line, which is typically 64 bytes on modern machines.

The Cache Hierarchy. The memory subsystem of modern CPUs often consists of a hierarchy of caches, which in a typical Intel CPU consists of three levels. Each core has two L1 caches, one for data and one for instruction, and one unified L2 cache. Additionally, the CPU has a last level cache (LLC), which is shared between all of the cores. When accessing memory, the processor first checks if the data is in the L1. If it is not found, the search continues down the hierarchy. In many Intel CPUs, the LLC is inclusive, i.e. its contents are a superset of all off the L1 and L2 caches in the cores it serves.

Cache Attacks. Timing access to memory can reveal information on the status of the cache, giving rise to side-channel attacks, which extract information by monitoring the cache state. Cache-based side-channel attacks have been demonstrated against cryptographic schemes [2, 13, 18, 36, 40, 45, 54, 69] and other secret or sensitive data [22, 57, 59, 60, 68].

B. Speculative Execution

To further improve performance processors execute instructions out of order. That is, instructions are executed as soon as their data dependencies are satisfied, even if preceding instructions have not yet completed execution. In case of branches whose condition cannot be fully determined, the processor tries to predict the branch outcome based on its prior behavior and speculatively execute instructions in the predicted target. Finally, in case of a misprediction, speculatively executed instructions become transient [35]. In this case, the processor drops all results computed by incorrect transient execution and resumes execution from the correct target address.

The disclosure of the Spectre [30] and Meltdown [35] attacks demonstrated that, contrary to contemporary beliefs, transient execution can have severe security implications. While the processor disposes of results computed by transient instructions, the effects of transient execution on microarchitectural components, including caches, are not reversed. Transient execution attacks exploit this effect by first obtaining information by triggering incorrect transient execution, and then subsequently leak it using via microarchitectural channels. Since the initial discovery of Spectre [30] and Meltdown [35] many variants other transient execution attacks [9] have emerged, including variants of Spectre [29, 30, 33, 38] and of Meltdown [8, 10, 35, 37, 49, 56, 62, 63, 64].

C. Microarchitectural Attacks in Browsers

Going beyond attacks using native code, side channel techniques have also been demonstrated using code running in sandboxed browser environments. Here, browser-based cache attacks have been used to classify user activity [44, 59, 60], and even extract cryptographic keys [17]. Attacks exploiting abnormal timings on denormal floating point values have been used for pixel stealing attacks [3, 31, 32] while Rowhammer-induced bit flips were also demonstrated using browser-based code [14, 16, 23]. Finally, browser-based transient-execution attacks using JavaScript code have also been demonstrated [30, 39], albeit without end-to-end exploits.

Eviction Set Construction. Cache attacks in JavaScript often require the ability to evict a value out of the cache. However, absence of native functionalities like `clflush` requires the attacker to utilize the cache architecture for evictions. For L1-D cache, eviction set construction can be done using only page offsets, i.e. if two elements having same page offsets belong to the same eviction set. However, as L3 caches have not so simple way to map an element to an eviction set, a more sophisitcated approach is required. Vila et al. [65] describes an approach to generate L3 eviction sets in Chrome browser.

Leaky.Page. Google has recently released `leaky.page`, a JavaScript-based Spectre Proof-of-Concept (PoC) which demonstrates recovering out of bounds information using Spectre v1 techniques [20]. More specifically, leaky.page first locates an instance of a `TypedArray` JavaScript object, whose length information and data pointer reside in different cache lines. It then evicts the array's length from the L1 cache, forcing speculation past the array length check upon array access. At the same time, as the cache line containing the array's data pointer is not flushed and remains cached, the attacker is able to perform a transient out-of-bounds array access, leaking the obtained data via a cache channel.

We note however, that JavaScript use of 32-bit array indices limits the effectiveness of leaky.page to the 4 GB heap containing `TypedArray` objects. This limitation is significant, as sensitive information (e.g., cookies, passwords, HTML DOM, etc.) is often located in different heaps and thus remains out of reach for Google's technique.

Finally, Google's PoC executes the Spectre v1 attack using a dedicated web-worker thread. While this reduces noise present on the cache side channel, web-worker threads use dedicated 32-bit heaps which are not shared with other websites rendered in the same process, further limiting the scope of data that can be obtained using Google's attack.

D. Strict Site Isolation

The ever increasing complexity of the internet has forced major design changes in modern browsers. Rather than rendering websites in a single monolithic process, browsers have come to adopt a multi-process architecture where multiple unprivileged rendering processes are used to render untrusted and potentially malicious webpages[1, 43, 50, 70]. In addition to the increased stability from crashes offered by this design, using unprivileged rendering processes limits an attacker's ability to mount an attack on the rest of the system by exploiting a vulnerability in the browser's rendering engine.

The Need for Greater Isolation. In an effort to strike a balance between performance and security, browsers may arbitrarily group the rendering of different websites into the same process. While grouping websites in this way provides some stability and security benefits, it provides little protection from attacks between websites sharing the same address space [5, 11, 12, 27, 53]. More recently, the introduction of transient execution attacks [30, 35] further highlighted the limitations of this approach, as these inherently challenge the notion of using software-based isolation for mutually-distrusting security domains. In particular, Spectre-type attacks have been demonstrated in browser contexts [30, 38, 63], allowing adversaries to dump the contents of entire address spaces, without the need to exploit any browser vulnerabilities.

Site Isolation. The next step in the evolution of browser architecture, comes in the form of strict site isolation [51]. Here, rather than arbitrarily grouping webpages into the same rendering process, strict site isolation aims to group webpages based on the location they are served from. In particular, strict site isolation uses the effective top-level domain plus one sub-domain (eTLD+1) as the definition of a security boundary, and ensures that multiple webpages can not be rendered by the same process unless they are all served from locations that share the same eTLD+1.

For example, Chrome will separate `example.com` and `example.net` as their top-level-domains, `.net` and `.com`, are different. `example.com` and `attacker.com` are also separated into different processes due to a difference in their first sub-domains (`example` and `attacker`). Finally, `store.example.com` and `corporate.example.com` are allowed to share the same process since they both share the same eTLD+1 `example.com`.

Chrome's Process Consolidation. Although two websites that share the same eTLD+1 domain might be consolidated into the same rendering process, Chrome may not immediately perform this consolidation. More specifically, Chrome aggressively consolidate websites that are embedded into each other as iframes, provided they share the same eTLD+1 domain. However, websites sharing eTLD+1 domain loaded into different tabs are only consolidated when memory pressure on the system crosses certain threshold, see Section III-A for a more complete discussion. Finally, Chrome's process consolidation is not only limited to websites, but also includes mutually-distrusting extensions installed on the machine. See Section V.

E. Chrome's Address Space Organization

Despite being a 64-bit application, Chrome still uses 32-bits to represent object pointers and array indices. More specifically, array indices are viewed as a 32-bit offset from the array's starting address, while 32-bit pointers represent the offset from a fixed base address in memory, which is often termed as the object's heap. Although this pointer compression design adds an additional layer of complexity, it does obtain a 50% saving in size of most pointers and array indices, resulting in a 20% saving in Chrome's overall memory consumption [58].

As a consequence of this optimization however, Chrome is limited to allocating objects within a span of 4 GB, referred by Chrome as partitions. Next, as a single 4GB partition is not sufficient to satisfy the memory consumption of resource heavy websites, Chrome uses multiple partitions based on the type of object being allocated. Finally, Google claims that this partition design also benefits browser security, as linear overflows can not corrupt information outside of their corresponding partition [19].

Chrome Heap Management. Chrome organizes the data corresponding to a website's content in multiple partitions (or heaps) [19], based on the content type. While many heaps exists, in this paper we focus on two heaps of particular interest. The first is the `node-partition` heap which holds the HTML nodes of the website being rendered. Next, we are also interested in the `buffer-partition` heap, which holds many buffer-based JavaScript objects (e.g., strings, hash-map, arrays, etc.) allocated during the execution of the main thread of the website rendering process.

Chrome Object Layout. When allocating buffer-like objects, Chrome follows a two stage process. First, Chrome allocates the memory required for the buffer storing the object's content. Next, Chrome allocates a metadata structure, which holds a pointer to the object's content buffer (called the `back-pointer`), as well as additional information such as the object's `type-id` and buffer length. Finally, certain JavaScript data structures, such as `Uint8Arrays`, often have their metadata structure and content's buffer located in two different heaps. In this case, the object's `back-pointer` must be comprised of 64-bits, as it needs to point outside the partition holding the object's metadata. See Figure 1.

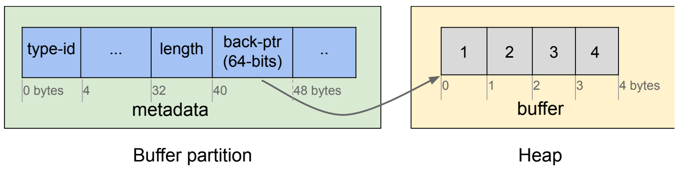


Figure 1: Representation of a Uint8Array array(1,2,3,4) object with data pointer in a different heap.

III. Spook.js: Mounting Speculative Execution Attacks in Chrome

We now present Spook.js, a JavaScript-based transient-execution attack that can recover information across security domains running concurrently in the Chrome browser. In addition to defeating all side-channel countermeasures deployed in Chrome (e.g., low-resolution timer), Spook.js overcomes several key challenges left open in previous works.

[$\mathcal{C}_1$] Strict Site Isolation. Strict site isolation prevents cross-site attacks. Website consolidation suggests an avenue for overcoming the challenge, but there is a need for a repeatable procedure that lets the attacker consolidate sites.

[$\mathcal{C}_2$] Array Index Limitations. JavaScript use of 32-bit array indices restricts bounds check bypass to only recover values from the same JavaScript heap. To retrieve all sensitive information, the attacker needs to be able to transiently access the full address space.

[$\mathcal{C}_3$] De-optimization. In our attacks, inducing mis-speculation also causes de-optimization that prevents multi-round attacks. For a successful attack, the attacker needs to cause mis-speculation without causing de-optimization.

[$\mathcal{C}_4$] Limited Speculation Window. Overcoming **[$\mathcal{C}_2$]** and **[$\mathcal{C}_3$]** requires a long speculation window. The attacker needs a consistent method that ensures that the processor does not detect mis-speculation before the transient code has the opportunity to retrieve the sensitive data and transmit it.

A. Overcoming [$\mathcal{C}_1$]*: Obtaining Address Space Consolidation*

To run the Spectre-type attacks, the attacker and target websites should reside in the same address space. Chrome's strict site isolation feature aims to prevent cross-domain attacks by segregating security domains into different address spaces. However, Chrome does consolidate websites with same eTLD+1 domain into the same process (see Section II-D). We now discuss how an attacker can exploit this consolidation to achieve co-residency between the attacker and target pages.

Exploiting iframes. We first observe it is possible to achieve consolidation by embedding iframes containing sensitive content originating from the same eTLD+1 domain as the attacker's page. For example, the attacker's page, `attacker.example.com` can contain an invisible iframe rendering content from `accounts.example.com`. If the targeted user is already logged in to `accounts.example.com`, the embedded iframe may contain sensitive personal information.

While effective, this method cannot operate on pages that refuse to be rendered inside embedded iframes, e.g. through setting `X-Frame-Options` to `deny`. Because many attacks exploit weaknesses in iframes, setting this option is a recommended security measure and is employed by many websites.

Obtaining Cross Tab Consolidation. In case of memory pressure, Chrome attempts to reduce its memory consumption by consolidating websites running in different tabs, provided that these have the same eTLD+1. Thus, a tab rendering `attacker.example.com` might be consolidated with another tab rendering `accounts.example.com`. Moreover, we observe that once consolidation occurs, Chrome tends to add newly-opened websites sharing the same eTLD+1 domain, rather then create a new process for them.

Abusing `window.open`. Next, Chrome often consolidates pages opened using the JavaScript `window.open(url)` API, when these have the same eTLD+1 domain as the opener. Thus, an attacker controlling `attacker.example.com` can use `window.open` to open a new tab containing `accounts.example.com`. While this method has the disadvantage of a new tab being visible, we find it to be particularly reliable as it does not require any memory pressure.

Experimental Results. We measure the effectiveness of each of the consolidation approaches described above using Chrome 89.0.4389 (latest at the time of writing) on a machine featuring an Intel i7-7600U CPU and 8 GB of RAM, running Ubuntu 20.04. We find that embedded iframes are always consolidated, if they have same eTLD+1 domain as the website embedding them. Likewise, we achieve perfectly reliable consolidation for the `windows.open` method described above.

To benchmark cross-tab consolidation under memory pressure, we simultaneously opened websites from Alexa's top US list, in different tabs. We have found that simultaneously opening 17 out of Alexa's top-20 websites forces Chrome to begin consolidation where possible. Finally, we find that the number of open websites depends on memory size. Specifically, on a similar machine with 16 GB, we need to open 33 sites concurrently before consolidation kicks in.

B. Overcoming [$\mathcal{C}_2$]*: Breaking 32-bit Boundaries via Speculative Type Confusion*

As described in Section II-E, Chrome uses a pointer compression technique that allows it to represent array indices and object pointers using 32-bit integers. In addition to 20% savings in memory consumption [58], this technique also achieves a security benefit, as linear buffer overflows cannot corrupt information outside of their 4 GB partition [19]. For an attacker trying to use JavaScript-based Spectre v1 techniques to read information outside of the allocated array size, Chrome's 32-bit representation seems to limit the scope of the recovered information to a single 4 GB heap, leaving the rest of the address space out of reach.

In this section, we show how to overcome this limitation using a novel type-confusion-based technique. Specifically, we construct a primitive that allows transient reading from an arbitrary 64-bit memory address chosen by the attacker. To that aim, we confuse the code that accesses arrays of type `Uint8Array` and cause it to operate transiently on an `AttackerClass` object that is under the attacker's control.

Before describing our speculative type confusion technique and the construction of our `AttackerClass` object, we now describe how Chrome accesses `Uint8Arrays`.

Performing Array Operations. Consider the following JavaScript array declaration `var arr = new Uint8Array(10)`. To implement `arr[index]`, Chrome JavaScript's engine performs the sequence of operations outlined in Listing 1. More specifically, Chrome first checks that the array type is `Uint8Array` (Line 3). If it is, Chrome verifies that `index` is within the array bounds (Line 9). Following the success of these checks, Chrome constructs

```
1  UInt8Array-access(array, index){
2    // type Check
3    if(array.type !== UInt8Array){
4      goto interpreter; // Wrong type
5    }
6    //compute array length
7    len = array.length
8    // length Check
9    if (index >= len) {
10       goto interpreter; // Out of Bounds
11   }
12   //compute array back_ptr
13   back_ptr = array.ext_ptr+
   ↪   ((array.base_ptr+heap_ptr)&0xFFFFFFFF);
14   //do memory access
15   return back_ptr[index];
16 }
```

Listing 1: Pseudocode of operations performed by Chrome's JavaScript engine during array accesses.

	Uint8Array	AttackerClass
1	type	type
2	properties	properties
3	integer_properties	integer_properties
4	buffer	f0
5	byte_offset_1	f1
6	byte_offset_2	f2
7	byte_length_1	f3
8	byte_length_2	f4
9	length_1	f5
10	length_2	f6
11	ext_ptr_1	f7
12	ext_ptr_2	f8
13	base_ptr	f9

Figure 2: Memory layout of objects of `Uint8Array` (left) and `AttackerClass` (right). All words are 4 bytes (32-bits). For clarity, 64-bit fields are split to two 32-bit words.

a pointer to the array backing store (Line 13), which it dereferences at offset `index` (Line 15). If either of the checks in Lines 3 or 9 fails, the execution engine raises an exception, diverting control to the JavaScript interpreter.

`Uint8Array` Memory Layout. The left half of Figure 2 shows the memory layout of a `Uint8Array` object. It starts with a three-field object header, specifing the object type and other properties. The header is followed by several fields, which describe the array. The important ones, from our perspective, are `length`, which specifies the number of elements in the array, and the two fields, `ext_ptr` and `base_ptr`, which are combined to get the pointer to the backing store. (The two fields are needed for legacy reasons.) We note that while `length` and `ext_ptr` are 64-bit wide, we present each of them as two 32-bit words in Figure 2 (left) for clarity.

A Malicious Memory Layout. Unlike published Spectre v1 attacks which exploit misprediction of a bounds check, our attack exploits type confusion by causing misprediction after a type check. For the attack, we cause transient execution of the `Uint8Array` array access code on an object other than a `Uint8Array`. By carefully aligning the fields of the malicious object, we can achieve transient accesses to arbitrary 64-bit memory locations.

The right side of Figure 2 contrasts the layout of the malicious object with that of `Uint8Array`. The `AttackerClass` object consists of ten 32-bit integer fields named `f0–f9`. We note, in particular, that `f5` and `f6` align with the `length` field of the array, `f7` and `f8` align with `ext_ptr`, and `f9` aligns with `base_ptr`.

Type Confusion. Type confusion operates by training the processor to predict that the object processed by Listing 1 is a `Uint8Array` and then calling the code with an `AttackerClass` object whose layout is as in Figure 2 (right). The crux of the attack is that during transient execution following a misprediction of the type check in Line 3 of Listing 1, the code accesses the fields of `AttackerClass` object but interprets them as fields of `Uint8Array`.

In particular, the code interprets the values of `f7–f9` as if they were `ext_ptr` and `base_ptr`. Thus, by controlling `f7–f9` an attacker can control the memory address accessed under speculation. Specifically, Line 13 of Listing 1 shows how the values of `ext_ptr` and `base_ptr` are combined with the global `heap_ptr` to calculate the pointer to the backing store. Because Chrome tends to align heaps to 4 GB boundaries, the 32 least significant bits of `heap_ptr` are typically all zero. Hence, by setting the values of `f7` and `f8` to the low and high words of the desired address, and setting `f9` to zero, the computed value of `back_ptr` is the desired address. If the value of `index` is also set to zero, the transient execution will result in accessing the desired access. Setting `index` to zero also helps avoiding the need to speculate over the test in Line 9 of Listing 1. All the attacker needs to do is to set `f5` and `f6` so that when they are interpreted as a 64-bit `length`, they yield a non-zero value, e.g., 1.

Delaying Type Resolution. Our type-confusion attack relies on mispredicting the branch at Line 3 of Listing 1. To allow transient-execution pass the branch, we need to delay the determination of the branch's condition. Typically, such delays can be achieved by evicting the data that the condition evaluates from the cache. In our case, we evict the `type` field of the `AttackerClass` object. At the same time to compute the fake backing store pointer, the CPU should have transient access to fields `f7–f9` of the `AttackerClass` object before the type-checking branch is resolved. Thus, for a successful attack, we need an `AttackerClass` object that straddles two cache lines. In Section III-D we describe how to achieve this layout and how we evict the `type` from the cache.

C. Overcoming [$\mathcal{C}_3$]*: Avoiding Deoptimization Events via Speculation*

In the previous section we show how we can perform type confusion with the compiler-generated code in Listing 1. We now demonstrate how an attacker can exploit type confusion

```
1  // Setup
2  let objArray = new Array(128);
3  for (let i=0; i<64; i++) {
4    objArray[i] = new Uint8Array(0x20);
5    objArray[64+i] = new AttackerClass(i);
6  }
7
8  let malIndex = findSplitAttackObj(objArray);
9  let malObject = objArray[malIndex];
10 let arguments;
11 let scratch = 0;
12 malObject.f0 = 1;
13
14 // Training
15 arguments = [0, 0];
16 for(let i=0; i<10000; i++) gadget();
17
18 // Attack
19 arguments = [malIndex, 0];
20 malObject.f5 = 1; // length_1
21 malObject.f6 = 0; // length_2
22 malObject.f7 = Lower32BitsOfAddress;
23 malObject.f8 = Upper32BitsOfAddress;
24 malObject.f9 = 0;
25 evict(malObject.f0);
26 gadget();
27 cacheChannel.receive()
28
29 // Attack Gadget
30 function gadget(){
31   let arrIndex, elemIndex = arguments;
32   if(arrIndex < malObject.f0) {
33     let arr = objArray[arrIndex];
34     let val = arr[elemIndex]; // byte
35     cacheChannel.leak(val);
36     return val;
37   }
38   return scratch;
39 }
```

Listing 2: An example of our speculative type confusion primitive, including code inserted by the JavaScript engine.

to construct a generic read primitive. The main complication is that the attacker needs to control not only speculative execution at the processor, but also ensure that the Chrome optimizing compiler does not modify the code as it runs.

Listing 2 presents the JavaScript code for the attack, which consists of four main stages that we now describe.

Setup. The attack relies on speculatively swapping a malicious object of `AttackerClass` for a `Uint8Array`. The setup stage prepares all the variables needed for the attack. It initializes an array of objects `objArray`, setting some of the entries to `Uint8Array` and others to objects of `AttackerClass` (Lines 1-6). In Line 7 it finds a malicious object that is split over two cache lines. (Recall from Section III-B that the attack requires such an object. We discuss the details of finding the object in Section III-D.)

We keep two different references to the malicious object. A direct reference in `malObject` (Line 9) and an indirect reference `malIndex`, via its index in `objArray`. Finally, the setup declares variables used by the gadget and sets `malObject.f0` to 1. We note that we assume that the field `f0` is in the same cache line as the malicious object's type.

Training. In Lines 15-16 we perform the attack's training stage. Here, we first set the arguments of the gadget to ensure that the accessed object is a `Uint8Array`. Specifically, the gadget expects two values in the variable `arguments`.[1] The first argument is an index to the array `objArray`, which can be either an object of `AttackerClass` or a `Uint8Array`. In the case that the pointed object is a `Uint8Array`, the gadget's second argument is the `Uint8Array` index. Setting the first argument to 0 implies that the gadget uses the object in `objArray[0]`, which is a `Uint8Array`.

After setting the arguments, the training stage invokes the gadget 10000 times (Line 16). During these invocations, Chrome's optimizer observes the gadget's execution, and detects that it always processes a `Uint8Array` object. Consequently, the optimizer specializes the gadget's array access to that case, using Listing 1 to perform the array access in Line 34. Moreover, the CPU's branch predictor observes the branches in the gadget and in the array access code, and sets their prediction to match a valid `Uint8Array` object.

Attack. In the attack stage, we first set the arguments to refer to our malicious object (Line 19). We then set the fields `f5`–`f9`, which correspond to the `length`, `ext_ptr`, and `base_ptr` of a `Uint8Array` (see Figure 2). Specifically we set `f5`, `f6` to have the value 1 when interpreted as array length, `f7`, `f8` to point to the desired 64-bit address when interpreted as `ext_ptr`, and `f9` which is interpreted as `base_ptr` to zero. We then evict the cache line that contains `f0` (and the malicious object's type) from the cache and invoke the gadget. When the gadget returns, we retrieve the leaked value from the cache side channel (Line 27), completing the attack.

Attack Gadget. The core of the attack occurs when the `gadget` function is executed on a `malObject` of type `AttackerClass`, after being specialized to handle `Uint8Arrays`. As Line 19 of Listing 2 passes `malIndex`, Line 33 results in `arr` being `malObject` of type `AttackerClass`. Next, as the array access in Line 34 of Listing 2 is specialized to handle access to `Uint8Arrays`, executing this line using `malObject` triggers the speculative type confusion attack described in Section III-B. In particular, the address encoded in `malObject.f7` and `malObject.f8` is dereferenced, resulting in `val` being populated with that address' contents. Finally, Line 35 transmits `val` via a cache side channel, where it is recovered in Line 27.

Deoptimization Hazard. Recall that the compiler specializes the array access in Line 34 of Listing 2 based on the observing `Uint8Array` accesses during the training phase. When the type confusion attack executes, the type check (Line 3 of Listing 1) recognizes the mismatch between `Uint8Array` and `AttackerClass`, aborts the specialized code, and alerts Chrome that other types may be used. Consequently, the Chrome deoptimizes the array reference, revoking the specialization. Unfortunately, unspecialized code is not vulnerable to our type-confusion attack, preventing subsequent iterations.

[1]We pass the arguments using a global variable because we find that using function parameters increases the noise in the cache side channel which we use to retrieve the leaked values.

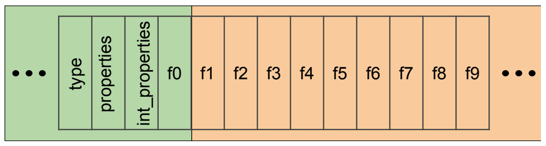


Figure 3: A malicious object split across cache lines.

Avoiding Deoptimization. To overcome this challenge, we adopt the *exception supression* technique of Meltdown [35]. Specifically, we wrap the attack code in a conditional block (Line 32 of Listing 2). The condition checks that the object index is less than the value of the field `f0` of the malicious object, which we have previously set to 1 (Line 12). Hence, the attack code is only executed architecturally when the index is zero and the refered object is `Uint8Array`. However, during the attack stage, the branch predictor mispredicts the condition as true, inducing a speculative execution of the entire type-confusion attack code, including the type check. Moreover, because evaluating the condition depends on the value of `f0`, which has been evicted from the cache (Line 25), the processor cannot detect the misprediction until after the type-confusion attack completes. When the procesor does detect the misprediction, it reverts the architectural state resulting from executing Lines 32-37, and proceeds to Line 38. In this case, the type mismatch only happens transiently and under speculation, and is never commit to the CPU's architectural state. Thus, Chrome is never alerted to mismatch and does not deoptimize the code in Listing 1 and Listing 2.

D. Overcoming [$\mathcal{C}_4$]*: Obtaining Deep Speculation via L3 Evictions*

The attack described in Listing 2 requires a malicious object of type `AttackerClass` whose memory layout straddles two cache lines (see Figure 3). In addition, the technique in Section III-C also requires the ability to evict `malObject`'s type from the CPU's cache while keeping fields `f1–f9` inside the cache. We now describe our technique for finding a malicious object at the desired cache layout, as well as how to flush its `type` ID from the cache. At a high level, we first find eviction sets for all of the LLC sets and then use these to locate an appropriate malicious object and evict its type.

```
1  LLCEvictionSets = GenerateLLCEvictionSets();
2
3  for(let i=64; i<128; i++){
4    candidate = objectList[i];
5    for(evictionSet in LLCEvictionSets){
6      evictionSet.evict();
7      let m = isEvicted(candidate.f0);
8      evictionSet.evict();
9      let h = isEvicted(candidate.f1);
10     if(m && !h){
11       malIndex = i;
12       break;
13 } } }
```

Listing 3: Finding an `AttackerClass` whose memory layout straddles two cache lines.

Finding a Split Object and a Corresponding Eviction Set. Listing 3 shows the code for identifying an appropriate malicious object and a matching eviction set that evicts the object type. The code first generates eviction sets for all of the cache sets in the LLC, using the approach of Vila et al. [65] (Line 1). It then tests each of the `AttackerClass` objects generated in Line 5 of Listing 2 to see if any has the desired layout. For each candidate, the test iterates over all of the eviction sets (Line 5), testing whether the eviction set evicts the candidate's `f0` but not its `f1` field. When an appropriate `AttackerClass` object is found, the code records its index in `malIndex` (Line 12), to be used in the attack of Listing 2.

Eviction Test. Testing whether a field has been evicted is done by measuring the time to access it. However, as part of hardening the browser against microarchitectural attacks, Chrome has reduced the resolution of its timer API [48]. Instead of using nominal APIs, such as `performance.now()`, we use the approach of Schwarz et al. [55], which exploits a counting thread with a `SharedArrayBuffer`. We note that following the discovery of Spectre [4], Chrome disabled the `SharedArrayBuffer` API [4], but ironically re-enabled them after strict site isolation has been deployed.

The Need For LLC Eviction. Rather than using LLC eviction sets, the Google PoC, Leaky.Page [20], uses L1 evictions. Constructing L1 eviction sets is by far simpler and is more reliable than constructing LLC eviction sets, raising the question of why not use the simpler approach for our attack.

Testing L1 evictions, we find that while these are good for standard Spectre attacks, L1 evictions are not suitable for our attack. More specifically, we observe that our attack consistently fails if we evict `malObject.f0` from the L1 cache but not from the LLC cache. We believe that the cause is related to the length of the speculation window (i.e., the number of instructions that the processor executes before detecting misspeculation). L1 misses are typically served from the L2, taking about 10 cycles. LLC misses, however, are served from the main memory, requiring over 100 cycles. We conjecture that our attack needs to speculatively execute more instructions that fit within the speculation window provided by L1 misses, requiring the longer, more complex, LLC misses.

E. End-to-End Attack Performance

So far, we have describe a combination of techniques that enable an attacker's webpage to recover the contents of any address in its rendering process. We now evaluate the effectiveness of our techniques across several generations of processors, including CPUs made by Intel, AMD and Apple.

Attack Setup. On Intel and Apple processors, we run Spook.js on an unmodified Chrome 89.0.4389.114, the latest version at the time of writing. For our benchmark, we initialize a 10 KB memory region with a known (random) content and then use Spook.js to leak its contents. To evaluate the speculative execution attack on AMD processors, where we could not generate effective eviction, we instrument V8 to expose the `clflush` instruction to JavaScript code, evaluating the attack under the assumption that an attacker can force eviction.

Eviction Set Based Results. Table I shows a summary of our findings, averaging over 20 attack attempts. As can be seen, Spook.js leaks 500 B/sec on Intel processors ranging from generation 6th to 9th, while maintaining above 96% accuracy rate. On the Apple M1, Spook.js achieves a leakage rate of 450 B/sec with 99% accuracy.

Processor	Architecture	Eviction Method	Leakage	Error
Apple M1	M1	Eviction Sets	451 B/s	0.99%
Intel i7 6700K	Skylake	Eviction Sets	533 B/s	0.32%
Intel i7 7600U	Kaby Lake	Eviction Sets	504 B/s	0.97%
Intel i5 8250U	Kaby Lake R	Eviction Sets	386 B/s	3.93%
Intel i7 8559U	Coffee Lake	Eviction Sets	579 B/s	1.84%
Intel i9 9900K	Coffee Lake R	Eviction Sets	488 B/s	3.76%
AMD TR 1800X	Zen 1	clflush	591 B/s	0.02%
AMD R5 4500U	Zen 2	clflush	590 B/s	0.06%
AMD R7 5800X	Zen 3	clflush	604 B/s	0.08%

Table I: Spook.js performance across different architectures.

Failing to Evict. Unfortunately, on AMD's Zen architecture we were unable to construct LLC eviction sets. As Spook.js requires the larger speculation window offered by LLC eviction, we were unable to run end-to-end Spook.js experiments on AMD systems. To evaluate the core speculative type confusion attack without constructing eviction sets, we instrumented V8 to expose the `clflush` instruction. As Table I shows, Spook.js achieves a rate of around 500 B/sec, demonstrating that if an efficient LLC eviction mechanism is found, Spook.js will be applicable to AMD. We leave the development of such eviction techniques to future work.

IV. Attack Scenarios

Having demonstrated the attack capabilities, we now turn our attention to its implications. In this section we investigate multiple real-world scenarios in which Spook.js retrieves secret or sensitive information.

Experimental Setup. We perform all of the experiments in this section using a ThinkPad X1 laptop equipped with an Intel i7-7600U CPU and running Ubuntu 18.04. we use an unmodified Chrome version 89.0.4389, which is the latest stable release at the time of writing. Finally, we leave all of Chrome's settings in their default configuration with all default countermeasures against side-channel attacks enabled.

Obtaining Consolidation. We recall the results from Section III-A, where Chrome consolidate websites into the same renderer processes based on their eTLD+1 domains. More specifically, as noted in Section III-A, Chrome will consolidate wbesites into the same renderer process either naturally due to tab pressure (33 tabs for 16GB machine) or due to the attacker opening the target page using the `window.open` API call.

A. Website Identification

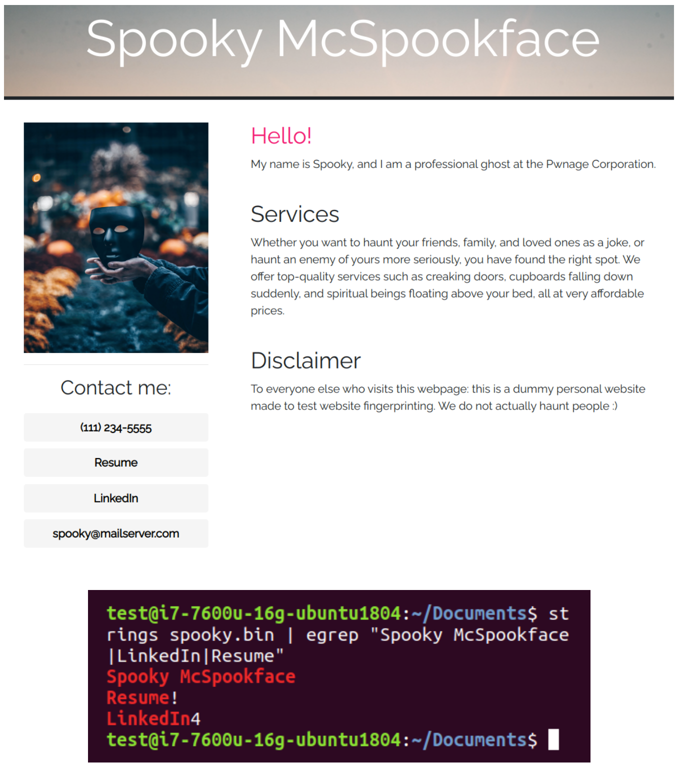
In our first scenario, we assume that our attacker has managed to upload a malicious webpage executing Spook.js attacking to a public hosting website, and managed to convince the victim to open his page on their machine. While most of the pages located on the hosting service are public, and thus have the contents of their DOM tree known to the attacker, information about what pages are being concurrently viewed by the user is private and is not attacker-accessible.

Attack Setup. We demonstrate how Spook.js violates this assumption, by creating a personal page on bitbucket.io, a Git-based hosting service. To mount our attack, we have hosted an attacker webpage on bitbucket.io which contains the code for mounting Spook.js. We have also created three sample personal pages on bitbucket.io, see Figure 4 (top). Following Bitbucket's naming convention, all four websites had URLs of the form of https://username.bitbucket.io/, making these eligible for consolidation. The ground truth usernames for our three sample personal pages were {`spectrevictim`, `lessknownattacker`, `knownattacker`}.

Experimental Results. After opening the four websites using four different tabs, we obtained consolidation of all four Bitbucket pages in one renderer process using the technique from Section III-A. We then used Spook.js to leak the memory space of the renderer process. Inspecting the result, we were able to recover a list of the URLs for the tabs being rendered, see Figure 5. Finally, while the contents of our sample victim pages are public, the list and contents of bitbucket.io websites simultaneously viewed by the user is private, and should not be accessible to a malicious page hosted on bitbucket.io.

B. Recovering Sensitive DOM Information

Having demonstrated the ability to recover DOM content, we now consider a scenario where the target subdomain is protected by an authentication requirement, and presents private


Spooky McSpookface

Hello!

My name is Spooky, and I am a professional ghost at the Pwnage Corporation.

Services

Whether you want to haunt your friends, family, and loved ones as a joke, or haunt an enemy of yours more seriously, you have found the right spot. We offer top-quality services such as creaking doors, cupboards falling down suddenly, and spiritual beings floating above your bed, all at very affordable prices.

Disclaimer

To everyone else who visits this webpage: this is a dummy personal website made to test website fingerprinting. We do not actually haunt people :)

Contact me:

(111) 234-5555

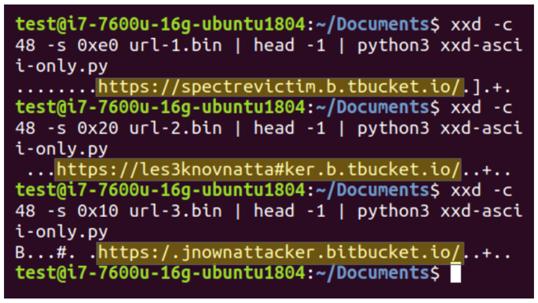
Resume

LinkedIn

spooky@mailserver.com

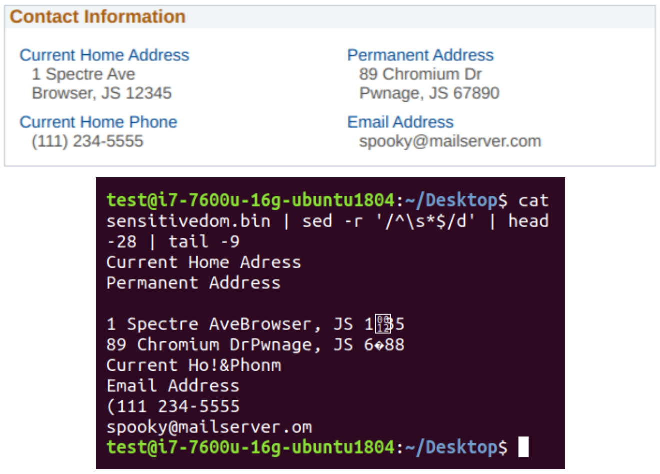
```
test@i7-7600u-16g-ubuntu1804:~/Documents$ st
rings spooky.bin | egrep "Spooky McSpookface
|LinkedIn|Resume"
Spooky McSpookface
Resume!
LinkedIn4
test@i7-7600u-16g-ubuntu1804:~/Documents$
```

Figure 4: (top) Example of a victim webpage. (bottom) Leakage of parts of the victim webpage's text.



```
test@i7-7600u-16g-ubuntu1804:~/Documents$ xxd -c
48 -s 0xe0 url-1.bin | head -1 | python3 xxd-asci
i-only.py
........https://spectrevictim.b.tbucket.io/.].+.
test@i7-7600u-16g-ubuntu1804:~/Documents$ xxd -c
48 -s 0x20 url-2.bin | head -1 | python3 xxd-asci
i-only.py
 ...https://les3knovnatta#ker.b.tbucket.io/..+..
test@i7-7600u-16g-ubuntu1804:~/Documents$ xxd -c
48 -s 0x10 url-3.bin | head -1 | python3 xxd-asci
i-only.py
B...#. .https:/.jnownattacker.bitbucket.io/..+..
test@i7-7600u-16g-ubuntu1804:~/Documents$
```

Figure 5: Leakage of currently open bitbucket.io subdomains. The parts corresponding to the URLs are highlighted.



Contact Information

Current Home Address 1 Spectre Ave Browser, JS 12345	Permanent Address 89 Chromium Dr Pwnage, JS 67890
Current Home Phone (111) 234-5555	Email Address spooky@mailserver.com

```
test@i7-7600u-16g-ubuntu1804:~/Desktop$ cat
sensitivedom.bin | sed -r '/^\s*$/d' | head
-28 | tail -9
Current Home Adress
Permanent Address

1 Spectre AveBrowser, JS 1�5
89 Chromium DrPwnage, JS 6�88
Current Ho!&Phonm
Email Address
(111 234-5555
spooky@mailserver.om
test@i7-7600u-16g-ubuntu1804:~/Desktop$
```

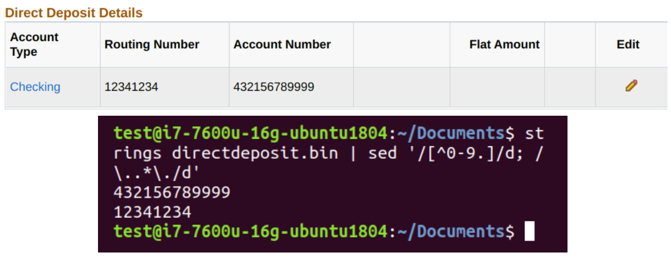
Figure 6: (top) Contact information page displayed of the university website, edited to show anonymized information. (bottom) Leakage of contact information using Spook.js.

data to the logged-in user. Here, we exploited the subdomain structure of the website of the University of Michigan, which at the time of writing hosts its main page, single sign-on (SSO) page, and internal portal page on the same eTLD+1 domain as personal webpages.

Attack Setup. Coordinating with the University's IT department, we hosted code performing Spook.js on a personal webpage (e.g., https://web.dpt.uni.edu/~user/).[2] With the author's account logged in, we visited three pages in the internal human-resources portal, https://portal.uni.edu/[2], on separate tabs. Each page contained the author's contact information, and direct deposit account. See Figure 6 (top) and Figure 7 (top), whose DOM was edited locally before mounting our attack in order to protect the authors' privacy.

Experimental Results. After opening the three tabs, we have also opened the page hosting Spook.js in the same window. Following its eTLD+1 consolidation policy, Chrome proceed to consolidate all four tabs into the same address space, allowing us to read sensitive values directly from the process' address space. See Figure 6 (bottom) and Figure 7 (bottom).

[2]Anonymized for submission

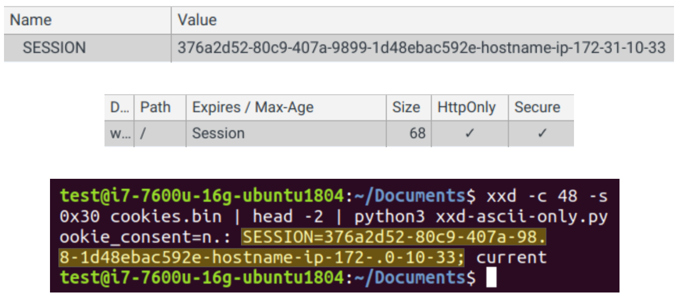


Direct Deposit Details

Account Type	Routing Number	Account Number		Flat Amount		Edit
Checking	12341234	432156789999				✎

```
test@i7-7600u-16g-ubuntu1804:~/Documents$ st
rings directdeposit.bin | sed '/[^0-9.]/d; /
\..*\./d'
432156789999
12341234
test@i7-7600u-16g-ubuntu1804:~/Documents$
```

Figure 7: (top) The direct deposit settings page of the university website, edited to show anonymized information. (bottom) Leakage of bank account and routing number.



Name	Value
SESSION	376a2d52-80c9-407a-9899-1d48ebac592e-hostname-ip-172-31-10-33

D...	Path	Expires / Max-Age	Size	HttpOnly	Secure
w...	/	Session	68	✓	✓

```
test@i7-7600u-16g-ubuntu1804:~/Documents$ xxd -c 48 -s
0x30 cookies.bin | head -2 | python3 xxd-ascii-only.py
ookie_consent=n.: SESSION=376a2d52-80c9-407a-98.
8-1d48ebac592e-hostname-ip-172-.0-10-33; current
test@i7-7600u-16g-ubuntu1804:~/Documents$
```

Figure 8: (top) Session cookie for the University of Michigan's portal page displayed using Chrome's developer tools. (bottom) Leakage of the session cookie.

C. Reading Login Information

Going beyond information present on the document's DOM, we repeated our attack on the University of Michigan's website, but this time targeting session cookies.

Attack Setup. We replicated the previous scenario's two-tab setup, where the Spook.js page (https://web.dpt.uni.edu/~user/), and internal portal page (https://portal.uni.edu/, after logging in) were rendered using two different tabs but shared the same Chrome rendering process.

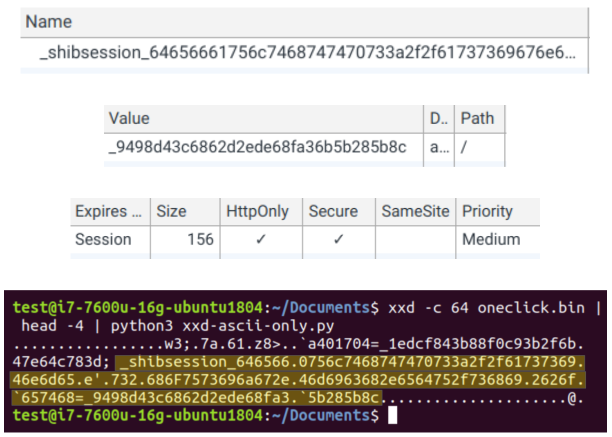
Experimental Results. Our initial approach was to locate the metadata of `document.cookie` object and dump the memory pointed to by its `back-pointer`, as shown in Figure 1. While we were able to read cookies associated with the portal page, the session cookie containing login information was marked as `HttpOnly`, and thus normally inaccessible from JavaScript. However, we discovered a different region in the address space of Chrome's rendering process that contains the session cookie, and successfully dumped it with Spook.js. See Figure 8. Finally, we note that our observations regarding `HttpOnly` cookies seems to contradict an explicit security goal of Chrome's strict site isolation, as the Google's strict site isolation paper explicitly states that `HttpOnly` cookies are not delivered to renderer processes [51, Section 5.1].

D. One-Click Cookie Recovery

The previous attack assumed a situation where the victim was authenticated to the University of Michigan, and had the attack code opened in a tab side-by-side with the university portal page. In this section we take a step further, showing that credentials can be sometimes recovered as soon as the target opened our malicious webpage, without the need to assume any simultaneously opened tabs.

Attack Setup. To that aim, we have added an iframe to our attacker's homepage (https://web.dpt.uni.edu/~user/), containing a webpage that displays private information and thus requires the user to be already authenticated in order to load. While the pages of most of the university's authenticated portals refused to load inside an iframe (presumably due to security reasons), this was not the case for the university housing portal, located at https://housing.uni.edu/. Here, in case a valid session cookie is present on the user's machine, meaning that the user is already authenticated), it will be used to login into the housing portal as it is being rendered inside an iframe on the attacker's page.

Experimental Results. We begin by recalling (Section III-A) that Chrome consolidates into the same renderer process a page and any of its iframes which share an eTLD+1 domain. Thus, after loading the attacker's page, Chrome immediately consolidated the address spaces of the attacker's webpage and the housing iframe. Next, as the user was previously authenticated through the university's single sign-on screen, the housing's portal iframe automatically accessed the university's `HttpOnly` login cookie, bringing it to the memory address space of the renderer process, without any user interaction. This, like in previous cases, allowed us to dump the login cookie, thereby bypassing the university's single sign-on screen. We show the ground-truth name and value and their leaked counterparts in Figure 9 (top) and (bottom).



Name
_shibsession_64656661756c7468747470733a2f2f61737369676e6...

Value	D..	Path
_9498d43c6862d2ede68fa36b5b285b8c	a...	/

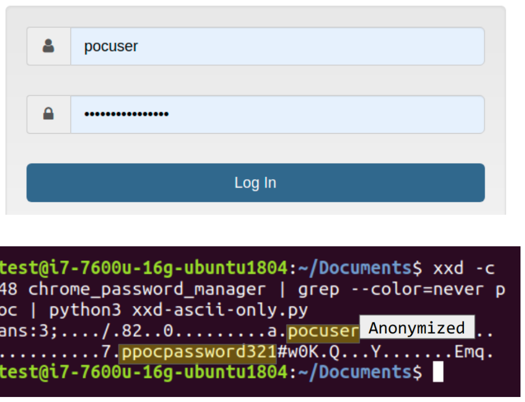
Expires ...	Size	HttpOnly	Secure	SameSite	Priority
Session	156	✓	✓		Medium

```
test@i7-7600u-16g-ubuntu1804:~/Documents$ xxd -c 64 oneclick.bin |
 head -4 | python3 xxd-ascii-only.py
....................w3;.7a.61.z8>..`a401704=_1edcf843b88f0c93b2f6b.
47e64c783d; _shibsession_646566.0756c7468747470733a2f2f61737369.
46e6d65.e'.732.686F7573696a672e.46d6963682e6564752f736869.2626f.
`657468=_9498d43c6862d2ede68fa3.`5b285b8c.....................@.
test@i7-7600u-16g-ubuntu1804:~/Documents$ 
```

Figure 9: (top) The housing portal's session cookie, loaded into the browser via an iframe. (bottom) Leakage of the housing portal's sesssion cookie from the one-click attack.

E. Attacking Credential Managers

We now demonstrate the security implications of Spook.js on popular credential managers, which auto populate the login credentials associated with a website, often without any user interaction. Moreover, we show that credentials can be recovered even without being submitted, namely without the user pressing the login (or any other) button, as merely populating the credentials into their corresponding fields brings them into the address space of the rendering process.



```
test@i7-7600u-16g-ubuntu1804:~/Documents$ xxd -c
48 chrome_password_manager | grep --color=never p
oc | python3 xxd-ascii-only.py
ans:3;..../.82..0.........a.pocuser Anonymized ..
..........7.ppocpassword321#w0K.Q...Y.......Emq.
test@i7-7600u-16g-ubuntu1804:~/Documents$ 
```

Figure 10: (top) Credential autofill by Chrome's password manager into the University of Michigan's login page. (bottom) leaked credentials using Spook.js.

Attacking Chrome. We utilize the previous two tab setup where the login page (https://weblogin.uni.edu/), and the internal attacker page (https://web.dpt.uni.edu/~user/) are hosted by our university and rendered by the same process. We assume that the credentials for https://weblogin.uni.edu/ are already saved with Chrome 89 password manager and it populates them as shown in Figure 10 (top). As shown in Figure 10 (bottom), Spook.js is capable of recovering the auto-populated credentials, without being submitted by the user.

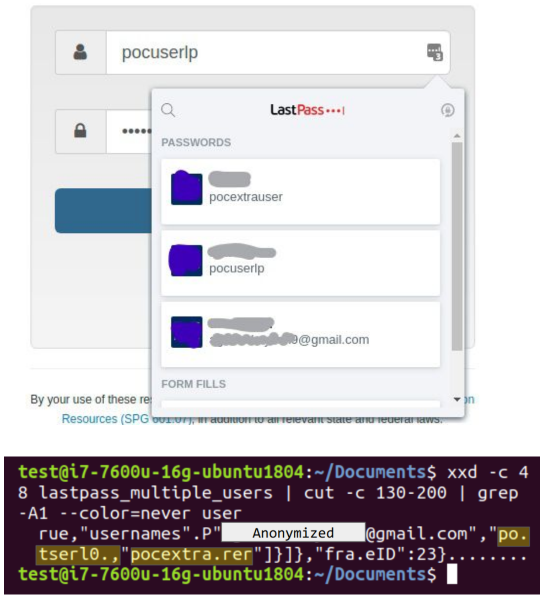
Attacking LastPass. Next, we used the same setup but this time using LastPass version 4.69.0 to autofill the passwords (instead of Chrome's password manager). In addition to obtaining similar password leaking results as in Figure 10, we were also able to get multiple account usernames associated with the website. More specifically, Figure 11 (top) shows multiple credentials that we associated with the University of Michigan's login site. As the list of accounts is present in the address space of the rendering process, it can be recovered using Spook.js. See Figure 11 (bottom).

Extracting Credit Cards. In addition to passwords, credential managers such as Lastpass can be used to manage credit card information, auto filling it when authorized by the user. Using a setup similar to attacking login credentials, we were able to read the victim's card information shown in Figure 12 (top) after it was populated on a payment page with the same eTLD+1 domain as the attacker's page. See Figure 12 (bottom). Finally, similar credit-card recovery results were also obtained when attacking Chrome's credit card auto-fill feature.

F. Exploiting Unintended Content Uploads

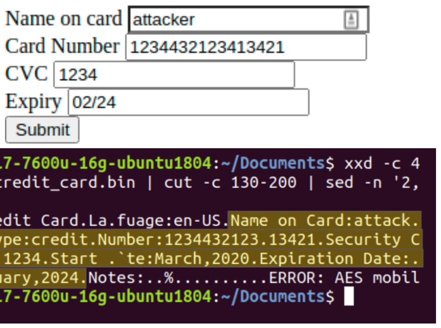
Our attacks so far assumed that the attacker's webpage is directly present on the domain to which the content was originally uploaded. We now depart from this assumption, showing that content uploaded to one domain is sometimes silently transferred to another, allowing Spook.js to recover it.

Google Sites. As an example of such a case, we use Google Sites, which allows users to create their own personal



```
test@i7-7600u-16g-ubuntu1804:~/Documents$ xxd -c 4
8 lastpass_multiple_users | cut -c 130-200 | grep
-A1 --color=never user
  rue,"usernames".P"   Anonymized   @gmail.com","po.
  tserl0.,"pocextra.rer"]}]},"fra.eID":23}........
test@i7-7600u-16g-ubuntu1804:~/Documents$
```

Figure 11: (top) Multiple accounts of the University of Michigan managed by LastPass. (bottom) Using Spook.js to leak the list of associated accounts.



Name on card	attacker
Card Number	1234432123413421
CVC	1234
Expiry	02/24

```
test@i7-7600u-16g-ubuntu1804:~/Documents$ xxd -c 4
8 lp_credit_card.bin | cut -c 130-200 | sed -n '2,
5p'
  :Credit Card.La.fuage:en-US.Name on Card:attack.
  r.Type:credit.Number:1234432123.13421.Security C
  ode:1234.Start .`te:March,2020.Expiration Date:.
  dbruary,2024.Notes:..%.........ERROR: AES mobil
test@i7-7600u-16g-ubuntu1804:~/Documents$
```

Figure 12: (top) Credit card information populated by LastPass. (bottom) Using Spook.js to leak credit card information.

webpage and embed HTML containing JavaScript under `https://sites.google.com/site/username/`. Google sites then run the user-supplied code in a sandboxed iframe, which hosts the code under `https://prefix.googleusercontent.com`. As we are unable to obtain a presence on the `google.com` domain, Spook.js cannot affect other `google.com` pages directly, as these are located on a different eTLD+1 domains and thus will never be consolidated with the attacker's page.

However, exploring Google services, we have observed that Google hosts more than personal webpages on `googleusercontent.com`. More specifically, Google seems to be using the `googleusercontent` domain as a storage location, automatically uploading emailed attachments, images, and thumbnails for Google-drive documents.

Google Photos. Focusing on Google Photos, we have discovered that all images uploaded (or automatically synchronized) to this service are actually hosted on the `googleusercontent.com`. When viewing images through `https://photos.google.com/`, the page loads the images from `googleusercontent.com` via `img` tags, which are unfortunately not consolidated into the process responsible for rendering pages hosted on `googleusercontent.com`.

Nevertheless, consolidation does occur in case the target elects for additional ways of viewing the image, via options available through the right-click menu. For example, if the target loads the image in a new tab, the image will come directly from `googleusercontent.com` making the tab eligible for consolidation. Likewise, in case the target opens the image through a link or QR code received from another person, the image will also rendered by a consolidatable `googleusercontent.com` tab.

Attack Setup and Experimental Results. We begin our attack by creating a webpage on `sites.google.com` that contains Spook.js attack code. Subsequently, similarly to previous experiments, we opened the attacker's Google site in one tab, and opened in a new tab rendering from the target's private Google Workspace (G Suite), which was automatically uploaded by Google on `googleusercontent.com`. See Figure 13 (left). After consolidation, we used Spook.js to recover the photo, see Figure 13 (right). While noise was present in the reconstructed image, the shape and color of the antelope were largely retained.

V. Exploiting Malicious Extensions

Moving away from security implications of website consolidation, in this section we look at the security implications of consolidating Chrome extensions. At a high level, Chrome allows users to install JavaScript-based extensions, that modify the browser's default behavior such as blocking ads, applying themes to websites, managing passwords, etc.

Extension Permissions. To assist in this task, Chrome uses a permissions model, providing extensions with capabilities beyond that of regular JavaScript code executed by a website. To secure these privileged capabilities from websites and other less privileged extensions, it is important that extensions are correctly isolated from each other and from websites.

The LastPass Extension. To demonstrate the security implications of consolidating Chrome extensions, we use the LastPass Chrome extension, which is a popular credential manager for syncing credentials across multiple devices belonging to

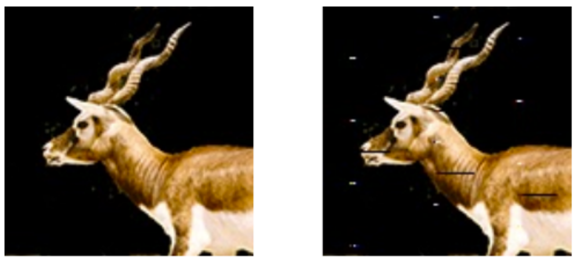
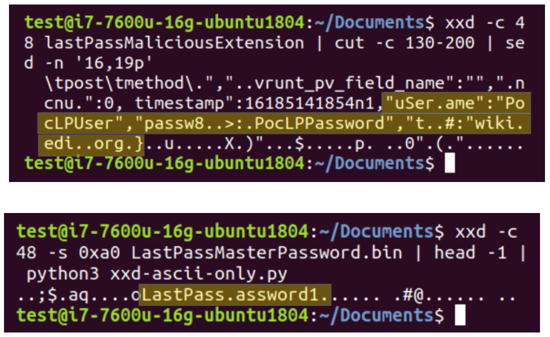


Figure 13: (left) An image of an antelope uploaded to Google Photos. (right) A reconstructed image from the leaked data.



```
test@i7-7600u-16g-ubuntu1804:~/Documents$ xxd -c 4
8 lastPassMaliciousExtension | cut -c 130-200 | se
d -n '16,19p'
  \tpost\tmethod\.","..vrunt_pv_field_name":"",".n
  cnu.":0, timestamp":16185141854n1,"uSer.ame":"Po
  cLPUser","passw8..>:.PocLPPassword","t..#:"wiki.
  edi..org.}..u.....X.)"...$.....p. ..0".(."......
test@i7-7600u-16g-ubuntu1804:~/Documents$
```

```
test@i7-7600u-16g-ubuntu1804:~/Documents$ xxd -c
48 -s 0xa0 LastPassMasterPassword.bin | head -1 |
 python3 xxd-ascii-only.py
..;$.aq....oLastPass.assword1...... .#@...... ..
test@i7-7600u-16g-ubuntu1804:~/Documents$
```

Figure 14: (top) Recovered login credential of wikipedia.org populated by LastPass. (bottom) Recovered master password of LatPass' vault (originally `LastPassPassword1#`).

the same user. When the user logs into Chrome's LastPass extension, the extension fetches the encrypted password stored on LastPass' cloud, and decrypts them using a key derived from the user's password [34]. Furthermore, we empirically observed that LastPass decrypts passwords only when it has to populate the credentials into a website, while retaining the all the usernames in plaintext in memory.

Attack Setup and Results. We use Chrome with the LastPass v4.69.0 extension, signing in into our LastPass account. We also ported Spook.js into a malicious Chrome extension that requires no permissions, and install it on our system. As the system was already under tab pressure (as described in Section III-A), we observed that Chrome had immediately consolidated our malicious extension with LastPass. We now use LastPass to login to any website, which triggers LastPass to decrypt and populate the website's credentials. Since Spook.js is running in the same process as the LastPass extension, we can leak the decrypted credentials thereby violating Chrome's extension security model. See Figure 14 (top). Going beyond credentials for specific websites, we were also able to leak the vault's master password, which allows the attacker to compromise all of the vault's accounts, see Figure 14 (bottom).

VI. Attacking Additional Browsers

We now move to investigating Spook.js on other Chromium-based browsers, namely Microsft Edge and Brave. Edge is the default browser shipped with Windows 10 (5% desktop market share [28]), whereas Brave is a popular privacy-oriented browser which aims to block ads and trackers [7]. As both of these browsers are based on Chromium, they inherit the strict site isolation policy and its security limitations.

We experimentally observe that the consolidation techniques of Section III-A are effective in both browsers. Measuring the effectiveness of Spook.js on both browsers, in Table II we see that, in both browsers, Spook.js achieves leakage rates similar to those obtained on Chrome.

Finally, we test the experimental implementation of strict site isolation on Firefox [41] using Firefox Nightly 89.0a1[42], build date April 12, 2021. Similarly to Chrome, we observe consolidation with tab pressure and `window.open`. However, due to significant JavaScript engine differences, we stop short of implementing Spook.js on Firefox.

Processor	Browser	Leakage Rate	Error Rate
Intel i7 6700k (Skylake)	Brave v89.1.22.71	504 B/s	1.25%
	Edge v89.0.774.76	381 B/s	4.88%

Table II: Spook.js performance on Brave and Edge

VII. Conclusions

In this paper we presented Spook.js, a new transient execution attack capable of recovering sensitive information despite Chrome's strict site isolation countermeasure. The fundamental weakness that Spook.js exploits is the differences in the security models of strict site isolation and the rest of the web ecosystem at large. On one hand, strict site isolation considers any two resources served from the same eTLD+1 to always be in the same security domain. On the other hand, the rest of the web enjoys a much richer definition of security domain that by default only considers two resources to be in the same security domain if the entire domain name is identical, often known as the same-origin policy. As we show in Section IV, the different definitions for a security domain has manifested as vulnerabilities in real world websites, that can be practically exploited by Spook.js.

A. *Countermeasures*

We discuss two mitigation strategies for Spook.js. The first aimed at website operators, who can adopt it to protect their users, including those of other browsers. The second strategy is aimed at browser vendors, suggesting changes to the definition of strict site isolation to better align with other security domains in the browser.

Separating User JavaScript. Spook.js relies on consolidating two endpoints of the same website into the same process, one which executes attacker controlled JavaScript and another which contains sensitive data. Website operators can protect their users from Spook.js by using different domains to serve each endpoint. While this technique is already used for separating user content from operator content, it is insufficient in cases where user-provided JavaScript is served from the same domain as other sensitive user-provided content. We propose to extend this idea, such that user-provided JavaScript content is served from one domain while all other user-provided content is served from another domain. This countermeasure can be immediately adopted by website operators to protect their users from JavaScript-based attacks such as Spook.js. While we expect Spook.js to be mitigated by vendors at the browser level, we still recommend adopting this separation as part of a larger defense-in-depth strategy.

Origin Isolation. For browser operators we propose aligning the definition of security domains in strict site isolation with those used by the rest of the web. A straightforward approach is to consider the entire domain name for strict site isolation, rather then relying on eTLD+1. Adopting this proposal will prevent consolidation of webpages from different subdomains, better matching the web's security model.

Performance Evaluation We evaluate the increased memory consumption of this approach by using Chrome's `--isolate-origins` feature, which allows users to supply a list of URLs to explicitly isolate. We visit each of Alexa's Top-50 websites individually and record Chrome's memory consumption with isolating all origins. We find that only 16% of them showed any increase in memory consumption at all. The worst case `yahoo.com`, resulted in two additional processes for an increase in memory consumption of 11%. We note that this naive evaluation misses many opportunities for process consolidation between concurrently opened tabs. To demonstrate this, we conduct an experiment where we consolidate `google.com` and `maps.google.com` into the same process. We find that consolidating tabs in this way can decrease memory consumption by as much as 30%. Therefore, measuring the real cost of this mitigation requires a user-study that measures consolidation rates in real-world browsing workloads. We leave the implementation of this mitigation and measurements of its cost to future work.

B. Limitations

Limitation of Targets. To deploy Spook.js, the attacker must be able to upload Spook.js JavaScript code to the target website's domain. While Section IV presents many such attack scenarios, Spook.js currently does not work across unrelated domains. Given the plethora of transient-execution attacks discovered and the complexity of modern browsers, it is not clear that unrelated domains are protected from each other.

Limitation of Architecture As described in Section III-E, we cannot mount end-to-end Spook.js on AMD systems due to our inability to reliably evict the machine's LLC cache. We leave the task of adapting the attack to the AMD Zen architecture to future work.

Attacking Firefox. Similarly to Chrome, Firefox's strict site isolation implementation also consolidates websites based on their eTLD+1 domain. While we successfully induced consolidation on Firefox, the JavaScript execution engine is significantly different from Chrome's. Thus, we leave the task of demonstrating Spook.js on Firefox to future work.

Acknowledgements

This work was supported the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; an ARC Discovery Early Career Researcher Award (project number DE200101577); an ARC Discovery Project (project number DP210102670); the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL) under contracts FA8750-19-C-0531 and HR001120C0087; Israel Science Foundation grants 702/16 and 703/16; the National Science Foundation under grant CNS-1954712; Len Blavatnik and the Blavatnik Family foundation and Blavatnik ICRC at Tel-Aviv University; the Research Center for Cyber Security at Tel-Aviv University established by the State of Israel, the Prime Minister's Office and Tel-Aviv University; and gifts from Intel and AMD.

References

[1] Webkit2. https://trac.webkit.org/wiki/WebKit2, 2011.

[2] Onur Acıiçmez. Yet another microarchitectural attack: Exploiting I-cache. In *CSAW*, pages 11–18, 2007.

[3] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *IEEE SP*, pages 623–639, 2015.

[4] Jake Archibald. Sharedarraybuffer updates in android chrome 88 and desktop chrome 91. https://developer.chrome.com/blog/enabling-shared-array-buffer/, 2021.

[5] Adam Barth, Joel Weinberger, and Dawn Song. Cross-origin JavaScript capability leaks: Detection, exploitation, and defense. In *USENIX security symposium*, pages 187–198, 2009.

[6] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*, 2017.

[7] Brave. Browse 3x faster than Chrome. https://brave.com/, 2021.

[8] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.

[9] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, pages 249–266, 2019.

[10] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *CCS*, pages 769–784, 2019.

[11] Shuo Chen, David Ross, and Yi-Min Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *CCS*, pages 2–11, 2007.

[12] Shuo Chen, Hong Chen, and Manuel Caballero. Residue objects: a challenge to web browser security. In *EuroSys*, pages 279–292, 2010.

[13] Shaanan Cohney, Andrew Kwong, Shahar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. Pseudorandom black swans: Cache attacks on CTR_DRBG. In *IEEE SP*, pages 1241–1258, 2020.

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

[15] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *CCS*, 2016.

[16] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the GPU. In *IEEE SP*, pages 195–210, 2018.

[17] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *ACNS*, pages 83–102, 2018.

[18] Daniel Genkin, Romain Poussier, Rui Qi Sim, Yuval Yarom, and Yuanjing Zhao. Cache vs. key-dependency: Side channeling an implementation of Pilsung. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(1):231–255, 2020.

[19] Google. Partition allocator. https://github.com/chromium/chromium/blob/master/base/allocator/partition_allocator/PartitionAlloc.md, 2021.

[20] Google. Spectre. https://leaky.page, 2021.

[21] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security*, pages 955–972, 2018.

[22] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, pages 897–912, 2015.

[23] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *DIMVA*, pages 300–321, 2016.

[24] Berk Gülmezoglu, Andreas Zankl, M. Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. Undermining user privacy on mobile devices using AI. In *AsiaCCS*, pages 214–227, 2019.

[25] Jann Horn. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[26] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE SP*, pages 191–205, 2013.

[27] Yaoqi Jia, Zheng Leong Chua, Hong Hu, Shuo Chen, Prateek Saxena, and Zhenkai Liang. "the web/local" boundary is fuzzy: A security study of Chrome's process-based sandboxing. In *CCS*, pages 791–804, 2016.

[28] Kinsta. Global desktop browser market share for 2021. https://kinsta.com/browser-market-share/, 2021.

[29] Vladimir Kiriansky and Carl A. Waldspurger. Speculative buffer overflows: Attacks and defenses. *CoRR*, abs/1807.03757, 2018.

[30] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE SP*, pages 1–19, 2019.

[31] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Security*, pages 463–480, 2016.

[32] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *USENIX Security*, pages 69–81, 2017.

[33] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *WOOT*, 2018.

[34] LastPass. How it works. https://www.lastpass.com/how-lastpass-works, 2021.

[35] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, pages 973–990, 2018.

[36] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, pages 605–622, 2015.

[37] Andrei Luțaș and Dan Luțaș. Bypassing KPTI using the speculative behavior of the SWAPGS instruction. In *BlackHat Europe*, 2019. URL https://i.blackhat.com/eu-19/Thursday/eu-19-Lutas-Bypassing-KPTI-Using-The-Speculative-Behavior-Of-The-SWAPGS-Instruction-wp.pdf.

[38] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *CCS*, pages 2109–2122, 2018.

[39] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR*, abs/1902.05178, 2019.

[40] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES*, pages 69–90, 2017.

[41] Mozilla. Project Fission. https://wiki.mozilla.org/Project_Fission, 2021.

[42] Mozilla. Firefox browser nightly. https://wiki.mozilla.org/Nightly, 2021.

[43] Nick Nguyen. The best Firefox ever. https://blog.mozilla.org/blog/2017/06/13/faster-better-firefox/, 2017.

[44] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *CCS*, pages 1406–1418, 2015.

[45] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, pages 1–20, 2006.

[46] Colin Percival. Cache missing for fun and profit. In *Proceedings of BSDCan*, 2005. URL https://www.daemonology.net/papers/htt.pdf.

[47] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security*, pages 565–581, 2016.

[48] Chromium Project. window.performance.now does not support submillisecond precision on Windows. https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110, 2016.

[49] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative data leaks across cores are real. In *IEEE SP*, 2021.

[50] Charles Reis, Adam Barth, and Carlos Pizano. Browser security: Lessons from google chrome: Google chrome developers focused on three key problems to shield the browser from attacks. *Queue*, 7(5):3–8, 2009.

[51] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: process separation for web sites within the browser. In *USENIX Security*, pages 1661–1678, 2019.

[52] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, pages 199–212, 2009.

[53] Roman Rogowski, Micah Morton, Forrest Li, Fabian Monrose, Kevin Z Snow, and Michalis Polychronakis. Revisiting browser security in the modern era: New data-only attacks and defenses. In *EuroS&P*, pages 366–381, 2017.

[54] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 lives of Bleichenbacher's CAT: new cache attacks on TLS implementations. In *IEEE SP*, pages 435–452, 2019.

[55] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *Financial Cryptography and Data Security*, pages 247–267, 2017.

[56] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, pages 753–768, 2019.

[57] Aria Shahverdi, Mahammad Shirinov, and Dana Dachman-Soled. Database reconstruction from noisy volumes: A cache side-channel attack on SQLite. *CoRR*, abs/2006.15007, 2020.

[58] Igor Sheludko and Santiago Aboy Solanes. Pointer compression in V8. https://v8.dev/blog/pointer-compression, 2020.

[59] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security*, pages 639–656, 2019.

[60] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses. In *USENIX Security*, 2021.

[61] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[62] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: hijacking transient execution through microarchitectural load value injection. In *IEEE SP*, pages 54–72, 2020.

[63] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue in-flight data load. In *IEEE SP*, pages 88–105, 2019.

[64] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. In *USENIX Security*, 2021.

[65] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. CacheQuery: Learning replacement policies from hardware caches. In *PLDI*, pages 519–532, 2020.

[66] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*, 2017.

[67] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE SP*, 2016.

[68] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Security*, pages 2003–2020, 2020.

[69] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, pages 719–732, 2014.

[70] Andy Zeigler. SharedArrayBuffer updates in Android Chrome 88 and desktop Chrome 91. https://docs.microsoft.com/en-us/archive/blogs/ie/ie8-and-loosely-coupled-ie-lcie, 2008.