On the Effectiveness of Same-Domain Memory Deduplication

Andreas Costi*
VU Amsterdam
Netherlands

Brian Johannesmeyer
VU Amsterdam
Netherlands

Erik Bosman
VU Amsterdam
Netherlands

Cristiano Giuffrida
VU Amsterdam
Netherlands

Herbert Bos
VU Amsterdam
Netherlands

ABSTRACT

Memory deduplication, an OS memory optimization technique that merges identical pages into a single Copy-on-Write (CoW) page, has been shown to be susceptible to a variety of timing side channel attacks, all of which stem from the differences between write times to the CoW page and to the normal page. To mitigate this issue, operating systems only merge pages *from the same security domain* (e.g., from the same process); moreover, browsers can piggyback on this defense with the recent adoption of site isolation. This was all considered sufficient, because it thwarts existing attacks, which have all relied upon separate domain (e.g., cross-process) scenarios.

In this paper, we examine the effectiveness of same-domain memory deduplication as a mitigation by presenting two case studies that show that an attacker can still leverage the deduplication side channel to leak secrets. Specifically, our case studies highlight one key flaw: that *it is non-trivial to separate programs into separate security domains*. In the first case study, we examine a client-server scenario—a scenario that inherently requires a server to read data from an untrusted client—and demonstrate that the client can control the alignment of data in memory to disclose the server's secret data. In the second case study, we examine a recent version of Firefox—a browser that has undergone massive efforts to ensure that data from different origins are separated into different domains—and demonstrate that nonetheless, a malicious webpage can exploit the browser's partial implementation of site isolation to leak secret data across tabs. We conclude that same-domain memory deduplication as a defense is difficult to implement correctly, and hence, is insufficient.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; *Browser security*.

KEYWORDS

memory deduplication, side channel attacks

*Also with KPMG Cyprus.

EUROSEC '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9255-6/22/04.
https://doi.org/10.1145/3517208.3523754

ACM Reference Format:
Andreas Costi, Brian Johannesmeyer, Erik Bosman, Cristiano Giuffrida, and Herbert Bos. 2022. On the Effectiveness of Same-Domain Memory Deduplication. In *15th European Workshop on Systems Security (EUROSEC '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3517208.3523754

1 INTRODUCTION

Memory deduplication is a memory optimization technique used by OSes and hypervisors [1, 2, 5] to scan and merge memory pages with the same content across processes and virtualized guest OSes. By keeping only one shared copy of a page, it reduces the total memory footprint of a running system, which is essential as many processes and guest OSes share memory pages with the same content (e.g., shared libraries, system files).

Due to the nature of memory deduplication, however, and the slow write operation on a deduplicated page, this mechanism inherently creates a timing side channel, which attackers [4, 22, 25] leveraged in the past to perform a variety of deduplication-based side channel attacks. To mitigate this, vendors either disabled deduplication entirely [32] (incurring a substantial performance penalty), or developed solutions to try to limit an attacker's capabilities. One solution in particular—to deduplicate pages based on a process's security context [15, 28, 35]—was adopted by Windows in the form of security domain-based memory deduplication [11]. Moreover, browser vendors both adopted site isolation, to piggyback on OS-based mitigations, and throttled native JavaScript timers, to hinder the ability to precisely time deduplication passes in the browser. Concurrent work [27] showcases how these mitigations can be bypassed, however, the focus of their work is on a specific type of attacker model (i.e., a remote attacker), whereas the focus of our work is on the mitigations themselves, while addressing the variety of attacker models that are possible.

In this paper, we present a detailed analysis of the applied mitigations and demonstrate that same-domain memory deduplication is insufficient because it assumes that programs judiciously separate trusted data and untrusted data into separate pages. However, we demonstrate that in practice—outside a few exceptions (e.g., programs that sanitize all untrusted input, browsers that implement full site isolation)—this assumption does not hold. In particular, we first present how in a client-server environment, a malicious client can exploit the server's single security domain by using an alignment probing primitive to perform byte-by-byte disclosure of the server's secret data by timing its responses. Furthermore, we present how, at the time of our research, Firefox's *partial* implementation of site isolation (which has since been fixed in Firefox

v94.0 [18]) could be exploited by a malicious webpage to achieve co-residency of separate tabs within a single security domain and that this, in combination with a custom timer, could be used to leak data across tabs.

Contributions. We make the following contributions:

- We present a detailed analysis of the mitigations deployed in recent OSes and browsers to combat memory deduplication side channel attacks and discuss how an attacker can still bypass them.
- We present two case studies[1] that bypass existing defenses—i.e., in (1) a client-server scenario and (2) a cross-tab browser scenario—highlighting the practical difficulty in adopting same-domain memory deduplication as a defense.
- We discuss other potential attack scenarios and potential mitigation strategies.

2 BACKGROUND

In this section we present the concept of *memory deduplication*, and discuss how deduplication creates a *timing side channel*. Further, we discuss mitigations applied on newer versions of Windows and in modern browsers, to limit an attacker's capabilities in launching deduplication-based side channel attacks.

2.1 Memory Deduplication

The OS or the hypervisor scans the physical memory, usually in predetermined time intervals, looks for pages with identical content, and remaps such pages to point to one of the same-content pages, making the rest of the pages free and available to the system. The deduplication mechanism updates the Page Table Entries (PTEs) of the owning processes to point to the shared copy of the page, and marks it as read-only to support CoW semantics. When a process that shares a deduplicated page tries to write to the page, a CoW page fault occurs at which point the system will create a private copy of the page and map it to the corresponding PTE of the faulting process. This CoW page fault leads to significantly slower write operations than a normal write to a non-deduplicated page, creating a *timing side channel* which an attacker can exploit, by crafting pages with the same content, wait for deduplication to merge same-content pages, and then write to one of the crafted pages, to detect if certain pages were already present in the system.

As illustrated in the past, this timing side channel provided attackers with the ability to fingerprint applications and websites [10, 23, 29–31] in hypervisors and in the context of a cross-VM sandboxed environment [7], perform information disclosure attacks to leak the address space layout of co-resident VMs in the cloud [3] and defeat Address Space Layout Randomization (ASLR), but also establish covert channels [33, 34].

More recent works, however, showcase that the deduplication side channel is quite powerful and leverage it in more complex cases. Flip Feng Shui (FFS) attacks [22, 25] use memory deduplication as a *massaging primitive*, and in combination with the Rowhammer vulnerability [8] (i.e., the ability to induce bit flips in memory pages by reading constantly from the physical memory), to corrupt the integrity of data stored in vulnerable to bit flips locations. The Dedup Est Machina [4] attacks describe how an attacker can use certain primitives (e.g., *alignment probing*, *birthday heap spray*), to leak code and heap pointers using JavaScript within Microsoft Edge, and how combining these two primitives with Rowhammer, an attacker can escape the browser's sandbox, and perform system-wide exploitation, by breaking ASLR from within the browser using JavaScript. Finally, concurrent work [27] show how a remote attacker can time HTTP/1 and HTTP/2 requests to a remote server and disclose memory contents, fingerprint system libraries' versions, leak database records, and defeat the KASLR of the virtual remote host.

[1]The implementations of our case studies are available at https://github.com/vusec/dedup-est-returns.

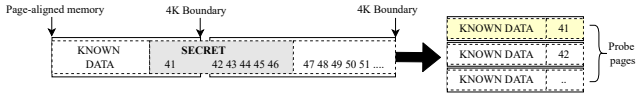
2.2 Windows Security Domains

To limit an attacker's capabilities in launching such advanced deduplication-based attacks, Microsoft introduced on newer versions of Windows (v.1903 onwards) *security domains* [12], where processes can choose to disable deduplication of their memory pages, via the `PROCESS_MITIGATION_SIDE_CHANNEL_ISOLATION_POLICY` structure, using the `DisablePageCombine` flag, and request the creation of a unique security domain with the `IsolateSecurityDomain` flag. This way a process can disable deduplication completely, even internally, except for *safe pages* (pages filled with 1's or 0's), whose content will not change during their lifetime. Unless a process explicitly declares the aforementioned, memory deduplication occurs on pages of processes within the same security domain. Two processes are eligible to belong in the same security domain if both processes effectively have full control over each other [14], and neither process explicitly requested its own security domain. Processes automatically inherit the security domain of their creator, unless the new process runs with a different process token, in which case this process will get its own security domain. A user-mode service periodically scans for processes that may be in a state where their security domains could be combined, and if compatible, combines them to create more deduplication opportunities.

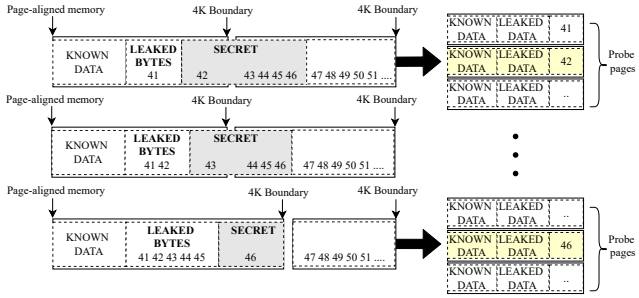
By not residing in the same security domain, malicious processes can no longer deduplicate their pages with pages of victim processes, thus limiting an attacker's capabilities to perform cross-process deduplication side channel attacks.

2.3 Browser Mitigations

Browser vendors introduced several mitigations to disable browser-based deduplication attacks. The most notable one was the throttling of JavaScript-based timers such as `performance.now()` [20]. To offer general protection, not only against deduplication side channel attacks, but also against other browser-based timing side channel and fingerprinting attacks, such as Spectre, browsers now round the returned value of `performance.now()` by some amount (typically to 1 millisecond increments), to be less predictable and reduce its accuracy. Firefox enables this by default, and can be configured to throttle the precision of native timers to either 1ms—which is the default—or 100ms. JavaScript code can utilize high precision timers only if the web documents are cross-origin isolated using the `COEP` [16] and `COOP` [17] policies. By requiring these headers, attackers cannot utilize other high precision timers (e.g., `SharedArrayBuffer` objects [21]), as attacker-controlled frames



(a) Layout of known data and secret data in memory.



(b) Incremental disclosure of secret data.

Figure 1: Alignment Probing primitive utilized to incrementally disclose a secret by manipulating the known data size.

(e.g., `iframe`) will not satisfy the required policies with victim pages. Finally, modern browsers enforce a per-process site isolation policy [13, 24] where the browser assigns each tab its own process. Each process receives its own security domain which as discussed previously disables inter-process memory deduplication. Firefox on the other hand, at the time of our research and prior to version 94.0 [18]—which implemented strict site isolation to mitigate against side channel attacks—used `N` content processes [19], where `N` tabs receive their own process, and hence unique security domain. When the browser spawns a `N+1` tab it results in sharing the same content process with another tab.

The adoption of these mitigations severely cripples browser-based deduplication attacks. Throttling the precision of timers disables attackers from accurately measuring deduplication passes and detecting when an attacker-controlled page is merged with a victim page. Further, per-process site isolation prevents inter-process memory deduplication, thus an attacker-controlled tab cannot deduplicate pages with a victim tab as their security domains will be different.

2.4 Attacker Primitives

Alignment Probing. We employ the same alignment probing primitive used in the Dedup Est Machina attacks [4], which provides an attacker with the ability to control the alignment of secret data based on the provided input. As such an attacker can incrementally leak secret data over several deduplication passes by manipulating the input provided to shift the secret data up and down in memory, and consequently in or out of the memory page. As a result, by pushing most of the secret (N-1 bytes) out of the memory page (Fig. 1a), the attacker can use deduplication to leak the first byte of the secret. The attacker repeats the process by providing a smaller (N-2 bytes) input, and including the first leaked byte of the secret in order to deduce the next byte (Fig. 1b), which allows the attacker to incrementally leak the whole secret over several deduplication passes.

SharedArrayBuffer timer. As mentioned previously, modern browsers severely crippled the precision (from $5\mu s$ to $20\mu s$) of native JavaScript timers (e.g., `performance.now()`), which prevented an attacker from accurately detecting deduplication passes. To measure accurately the write operation and detect the deduplication signal, an attacker can craft a custom JavaScript timer utilizing the `SharedArrayBuffer` JavaScript object.

`SharedArrayBuffer` allows two threads to share state. The first thread operates as the timer and the second thread reads the time. The timer thread utilizes `Atomics` to perform increment operations, and the reader thread can read the time at any point without the risk of a race condition. Using a `SharedArrayBuffer` timer, an attacker can still measure the time needed for an operation to complete. As such an attacker can poll the timer, perform a write operation and then poll the timer again in order to find out how long it takes to perform a write operation on a page. When the attacker sees a higher number of increments, it means that the deduplication thread combined the attacker-controlled page with a victim page.

3 THREAT MODELS

In this section we present our threat models including the attacker's capabilities with the purpose of leveraging the deduplication side channel to extract sensitive data from a server application and from a user tab in the browser. For both threat models we assume that page combining within the same security domain is enabled (enabled by default on Windows).

We emphasize that the following use cases represent two synthetic scenarios and represent a proof-of-concept in an attempt to showcase that same domain memory deduplication is difficult to implement correctly as it is perfectly normal—and often required—for processes to share memory and hence the same security domain.

Even though we focus on the following use cases, these two threat models represent classes of attack scenarios. In the browser environment, an attacker can leverage the ability to utilize a custom timer, and also leverage the ability to reside in the same content process as a victim tab, to perform fingerprinting attacks and retrieve sensitive information of the victim. In the client-server environment, server applications (e.g., nginx) that use in-memory key-value stores for storing arbitrary data (e.g., memcached), and that can be induced to create a desired page in their memory by a client application issuing targeted requests (e.g., API calls, etc.) are a potential target of deduplication side channel attacks.

Windows Native Client-Server Environment. For this type of attack we assume a Windows native cross-process environment running on Windows 10 x64 v2004, where two local processes (a client and a server implemented in `C++`) communicate via socket connections utilizing the `WinSock2` library. The attacker has full control over the client application, and is able to perform multiple requests to the server application. The purpose of the attacker is to time the requests to detect deduplication passes and incrementally leak a server's secret using an alignment probing primitive.

Browser Cross-Tab Environment. For this type of attack we assume a browser environment where a capable attacker can execute JavaScript code in the victim's browser, by either performing social engineering attacks, so the victim visits an attacker-controlled

website or by compromising a legitimate website. The attacker via the malicious website creates arbitrary memory pages containing synthetic fingerprints using `TypedArray` objects which are large enough to be page-aligned. We assume the victim visits legitimate websites over at least 8 tabs which load the corresponding fingerprints in memory. The purpose of the attacker is to detect deduplication passes by measuring the write time to the attacker-controlled pages, by utilizing a custom timer implemented using a `SharedArrayBuffer`, and hence detect which victim fingerprint was deduplicated with an attacker-controlled page. We further assume that the victim visits the malicious website for at least 15 minutes so the deduplication thread combines a victim fingerprint with an attacker-controlled page, that the `COOP` and `COEP` policies are enabled so that the `SharedArrayBuffer` custom timer works, and that a Firefox build prior to Firefox v94.0 is utilized which at the time of our research did not implement strict site isolation.

4 ATTACK VECTOR VALIDATION

As discussed in Section 2.1, deduplication inherently creates a timing side channel. This occurs because writing to deduplicated memory is quite slower than writing to normal memory. On Windows systems memory deduplication occurs every 15 minutes via a kernel thread which scans the memory for pages with identical content [6], given that the pages are eligible for merging based on their processes' security domains and their permissions. In this section we discuss native inter- and intra-process deduplication, and how deduplication-based side channel attacks are still feasible in the browser.

4.1 Same Security Domain Deduplication

With the introduction of security domains, the first step was to confirm if intra-process deduplication of arbitrary memory pages is still possible and if the deduplication signal is powerful enough to detect deduplication passes. To illustrate this over a native Windows process, we allocate a large number of 4KB pair-wise memory pages to be readable and writable (`PAGE_READWRITE`), and also be able to be deduplicated (`PAGE_WRITE COMBINE`). We fill them with arbitrary content (`0x41`, `0x42`, etc.) and measure how long it takes to write to a page, to create a baseline metric.

The average time needed for writing to a normal page in this scenario was $6.3\mu s$. After having this baseline metric, the process waits for a deduplication pass and then performs a write operation to the allocated memory pages to detect if there is a deduplication signal and if deduplication occurs on pages with arbitrary content. The average time needed for writing to a deduplicated page was $61.02\mu s$, which is significantly slower than a normal write operation. Figure 2 shows that over a sample size of 210 measurements, approximately 80% of the normal write operations took $6\mu s$. In contrast, the write operation on deduplicated pages takes more than $40\mu s$, showing that intra-process deduplication of arbitrary memory is still possible, and an attacker can successfully measure and detect slow write operations.

As stated in Section 2.2, processes with sufficient privileges over each other, can belong in the same security domain, which enables inter-process deduplication. Allocating arbitrary memory pages in the memory of a victim process and utilizing an attacker-controlled process to create and inject the pair memory pages in the victim process allows for the deduplication of attacker and victim pages. The average time needed for the attacker process to write normally to the allocated pages was $13.56\mu s$. After a deduplication pass the attacker process writes to the pair pages to determine if the deduplication thread merged any pages. The average time needed for the write operation to complete over attacker-controlled arbitrary pages was $67.56\mu s$, indicating that inter-process deduplication over the same security domain is feasible. Figure 2 shows that over 50 measurements 98% of normal write operations, in our inter-process scenario, took less than $30\mu s$ to complete. In contrast, 54% of the write operations on deduplicated pages took more than $80\mu s$ to complete.

Same security domain memory deduplication, however, will prevent us from deduplicating memory between an attacker-controlled client process, and a target server process in a different security domain. But it is perfectly normal—and often required—for processes from one security domain to read data into their address space from another (untrusted) security domain. This way, we can cause deduplication to happen within the victim's address space, by using attacker supplied data. By timing subsequent interactions with the victim process we can try to determine whether any CoWs occur on the server.

4.2 Browser Same-Tab Deduplication

An attacker can still create arbitrary memory pages via JavaScript, that are large enough to be page-aligned with `TypedArray` objects (discussed in Section 5.2.1). Further, utilizing custom timers [9, 26] or a `SharedArrayBuffer`, an attacker can create an accurate enough timer to measure write operations on the `TypedArray` objects and detect deduplication successfully. To test this in the browser, we create pairs of pages and write to pages that belong to different pairs every 10 seconds. We poll the timer (i.e., the reader thread) before performing the write operation, and again after concluding the write operation, to find out how many increments the timer thread performed.

The average time needed for a write operation to complete on normal pages is 333.82 "ticks", while writing to a combined page takes on average 5413.98 "ticks". As such, detecting deduplication in the browser utilizing a custom timer is still possible. Over a sample size of 139 measurements (Fig. 3), approximately 94% of the normal write operations took less than 1000 "ticks" to complete. In contrast, the write operation on deduplicated pages takes more than 1400 "ticks" to complete, and as such an attacker can successfully measure and detect slow write operations in the browser.

5 CASE STUDIES

In this section we present our two case studies where we showcase how deduplication is still feasible on Windows 10 native processes and in the browser in a cross-tab environment.

5.1 Client-Server Use Case

Knowing that the deduplication signal is quite powerful across processes (Section 4.1), we utilize this knowledge in our client-server attack. In a client-server environment, the server allocates memory and the client times how long it takes for the server to

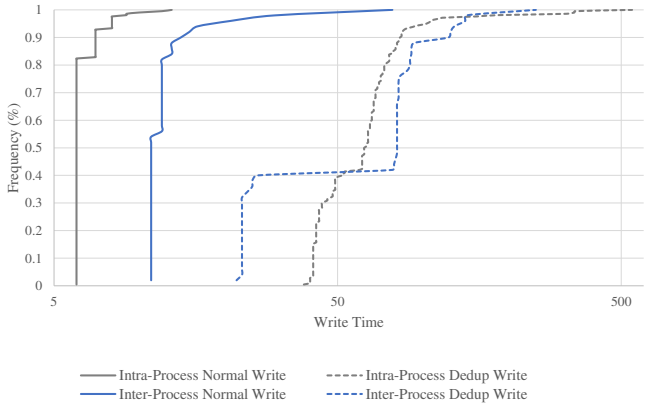


Figure 2: CDF of Intra-Process and Inter-Process write operations on normal and deduplicated pages.

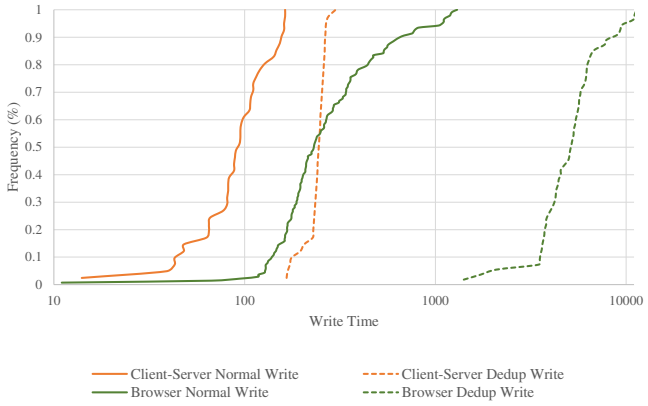


Figure 3: CDF of client-server and browser write operations on normal and deduplicated pages.

respond when writing to the allocated pages. Writing to a server's normal page and waiting for the server to respond is significantly faster than writing to a deduplicated server page and waiting for the server's response. This way, security domains do not play a part in limiting the deduplication side channel, since the attacker is not bound by deduplicating client and server memory pages.

5.1.1 Exploitation. For our client-server use case we utilize two local native processes which communicate via sockets. We assume that the client is attacker-controlled and can send multiple requests to the server at any given time. The server receives the client requests and allocates memory based on the attacker-provided input, whereby it stores the client input (hereby referred to as known data) at a page-aligned location next to secret data (Fig. 1a).

Knowing that the server stores the provided data next to secret data and that they are page-aligned, the attacker sends an initial request of 4095 known bytes. Further, due to the weak alignment properties of the server, the attacker is employed with the alignment probing primitive described in Section 2.4, which will result in the server creating the "secret" page containing the 4095 attacker-provided bytes and next to it will be the secret data. Due to the weak alignment properties, only the first byte of the secret will reside in the same page with the known data, which reduces the entropy of the secret, as the guesses required to leak a single byte are exponentially less than leaking the whole secret at the same time (Fig. 1a). The attacker sends multiple requests to the server containing the same 4095 bytes and 1 guessed byte at the end, spraying the memory of the server with "probe" pages. In this case, the server-allocated memory will contain only attacker-provided input. The attacker provides inputs to the server every 10 seconds in order to write to the "probe" pages, and measures how long it takes for the server to respond. When the response takes longer than a moving average, the attacker deduces that the deduplication mechanism combined a "probe" page with the "secret" page. The client then repeats the process by sending 1 less byte in the initial and probe requests and including in the known data the first leaked byte of the secret. As such the attacker performs byte-by-byte disclosure of the secret data over several deduplication passes depending on the length of the secret data (Fig. 1b). Assuming that an attacker wants to leak a 15-byte secret utilizing a 52-character alphabet, it will take 15 deduplication passes, 225 minutes, and 217MB of memory.

The average time needed for writing to a normal page, and for the server to respond was 96.73μs. When a deduplication pass occurs on the server, the time needed to perform the write operation and for the server to respond was 239.86μs, which is significantly slower. Figure 3 shows that in the client-server scenario, over a sample size of 41 measurements all normal write operations took less than 170μs. In contrast, 85% of the write operations on deduplicated pages took more than 205μs, allowing an attacker to detect deduplication by timing the server's responses.

Figure 4 shows the time needed for the write operations to complete on normal and deduplicated pages over the 6 deduplication rounds to fully disclose the secret data. Figure 1b illustrates the incremental disclosure of a 6-byte secret by utilizing the alignment probing primitive and the deduplication timing side channel to detect when deduplication occurs, combining a probe page with the secret page.

5.1.2 Limitations. A possible factor that could impair this attack is the server application being under load and hence affect the precision of measuring deduplication passes. A possible workaround to this is utilizing signal amplification techniques, whereby instead of measuring the time needed for a write operation to complete (i.e., the time needed for the server to respond) for 1 page, we are measuring the time needed for a write operation to complete for e.g., 100 pages so if several pages are deduplicated the deduplication signal is amplified.

5.2 Browser Cross-Tab Use Case

Deduplication-based attacks are still possible in a browser environment as discussed in Section 4.2. This only illustrates, however, that deduplication is feasible within the same browser tab which does not have a significant impact. Further, as discussed in Section 2.3, cross-tab deduplication-based attacks should not be possible, due to browsers enforcing site isolation where each tab resides in its own process, thus disabling deduplication of memory of two different tabs as they belong in different security domains.

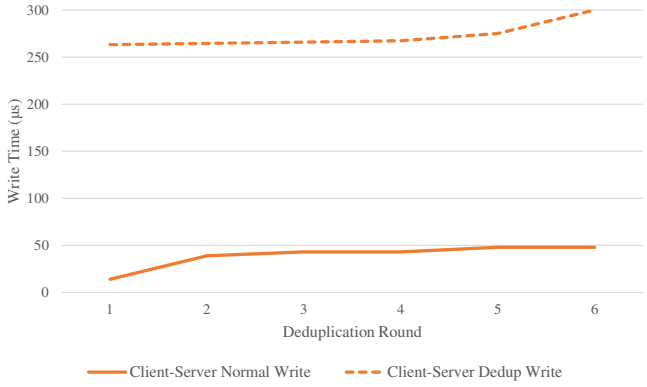


Figure 4: Write time on normal pages and deduplicated pages in a client-server scenario over 6 deduplication passes.

Firefox, however, at the time of our research did not yet enforce strict site isolation, and instead each tab resided in a Firefox content process. The maximum number of content processes available in Firefox was 8 and as such having 9 tabs open resulted in two tabs sharing the same content process, thus enabling deduplication, as the tabs share the same address space. This way an attacker-controlled tab in the victim's browser could share the same process with a victim tab, whereby the attacker could detect and force deduplication of attacker-controlled and victim pages.

5.2.1 JavaScript Deduplication. To successfully detect memory deduplication over JavaScript, an attacker must have control over how known data are stored in memory, and have an accurate timer to measure the write operations to detect deduplication passes. To satisfy the first requirement, an attacker can utilize `TypedArray` objects (e.g., `Uint8Array`), which when large enough are page-aligned in memory. This allows for controlling the content of each byte within the memory page, which subsequently provides an attacker with the ability of creating arbitrary memory pages. To satisfy the second requirement, we utilize a custom `SharedArrayBuffer` timer as described in Section 2.4.

5.2.2 Exploitation. We utilize one attacker tab and 8 victim tabs. We assume that a victim visits an attacker-controlled website and all 9 tabs remain open throughout the attack.

Firefox uses by default 8 content processes whereby each open tab runs its web content in one of these content processes. Only the first 8 open tabs, however, will reside in their own content process enforcing memory isolation. When a new tab opens, Firefox will arbitrarily assign it to one of the 8 content processes, which results in sharing memory with one of the 8 open tabs. As such, the attacker-controlled tab will reside in the same content process with a victim tab which also enables the deduplication of attacker-controlled memory and victim memory. By exploiting this limitation in content processes, it was possible to bypass the site isolation mitigation and force deduplication of attacker and victim memory. To bypass the timer limitation, we utilize a `SharedArrayBuffer` timer as discussed in Section 5.2.1. This way it was possible to create an accurate baseline for how long a write operation to a `TypedArray` object takes, and infer when deduplication occurs between an attacker-controlled and a victim page. In this use case we load 8 victim tabs whereby we encode multiple fingerprints, which are large enough to be page-aligned, in secret pages using `Uint8Array` objects. The attacker-controlled tab creates several probe pages using `Uint8Array` objects containing such fingerprints and waits in order to detect a deduplication pass. By encoding multiple fingerprints in secret pages and encoding fingerprints in probe pages, inevitably after a deduplication pass one of the secret and probe pages will be deduplicated, depending on which content process is shared between the victim and the attacker tabs. This way it was possible to leak a tab's fingerprint using the deduplication side channel in a browser cross-tab environment.

Due to the fact that the tabs' fingerprints are known, it was possible to leak a fingerprint in a single deduplication pass. By writing to all attacker-controlled pages which contain the fingerprints and looking at the time needed for a write operation to complete to detect which is significantly slower, we can infer which victim tab resides in the same content process as the attacker tab, and also which probe page was deduplicated with the victim's secret page which contains the tab's fingerprint.

6 DISCUSSION

In this section, we first discuss other potential attack scenarios that may bypass same-domain memory deduplication, address concurrent work, and discuss potential mitigation strategies.

Other attack scenarios. One possible high-value target of a memory deduplication side channel attack is the Window Registry, because it employs a single security domain while storing keys from a variety of security contexts (i.e., possibly from an attacker). However, due to restrictions in data mapping and accessing, and the fact that new Registry entries are not necessarily page-aligned and are stored in arbitrary locations in memory, deduplication-based attacks over Registry entries were not feasible. Further, arbitrary pairwise memory pages of processes that belong to different security domains, presented similar timings in write operations (11.45μs), meaning that across deduplication passes, the deduplication kernel thread did not combine them. Finally, during our research, related to our browser use case, we tested other browsers (e.g., Google Chrome and Microsoft Edge) to confirm if cross-tab deduplication is feasible in other browsers other than Firefox. These browsers, however, enabled by default strict site isolation—which Firefox implemented recently—thus disabling cross-tab deduplication, so only deduplication within the same tab is possible.

Concurrent work. As discussed in Section 2, concurrent work [27] focuses on how a remote attacker can time HTTP/1 and HTTP/2 requests to a remote server and disclose memory contents, fingerprint system libraries' versions, leak database records, and defeat the KASLR of the virtual remote host. Our work on the other hand focuses on the effectiveness of the deployed mitigations such as security domains and site isolation, and how attackers can bypass them to detect deduplication across processes with different security domains, and in the browser how by building a custom JavaScript

timer, attackers can still detect deduplication and perform fingerprinting attacks in a cross-tab scenario, thus addressing a variety of attacker models that are possible.

Mitigation Strategies. Possible mitigation strategies may include disabling deduplication completely, however, this nullifies all performance improvements deduplication offers. On the other hand, defense mechanisms such as VUsion [22] employ the same behavior on all pages of a system by removing all access permissions from pages that are considered for deduplication, which will always result in a copy-on-access page fault, thus disabling an attacker from distinguishing deduplication passes. Further, by taking advantage of the fact that deduplication is a slow background process and works mostly on the idle pages of a system, VUsion introduces a 2.7% performance overhead, making it a viable defense mechanism against the deduplication timing side channel.

7 CONCLUSIONS

In this paper we have shown that, even with all new system and browser mitigations applied, an attacker can still find ways to abuse the deduplication side channel and leak secrets in native Windows environments and in the browser. By utilizing the intra-process deduplication in a client-server environment, security domains do not play a part in mitigating deduplication-based attacks, as the attacker only needs to time the server's responses to detect deduplication. In the browser we have shown that a custom JavaScript timer still detects deduplication passes, defeating the timer throttling mitigation. Further, due to the fact that Firefox did not enforce strict site isolation, cross-tab deduplication was still possible, given that a large number of tabs are open in the victim's browser. As such, while the mitigation strategies discussed in Section 2 are quite effective in limiting an attacker's capabilities and attack surface, they are not completely effective against deduplication-based attacks as it is inherently difficult to implement same domain memory deduplication correctly, due to the fact that processes are often required—and is perfectly normal—to share memory and hence the same security domain.

Disclosure. We disclosed our findings to Microsoft on Jan 28, 2022. Firefox mitigated the issue we discovered prior to our paper submission.

ACKNOWLEDGEMENTS

We thank our shepherd, Alessandro Sorniotti, and the anonymous reviewers for their valuable comments. This work was supported by the EU's Horizon 2020 research and innovation programme under grant agreement No. 825377 (UNICORE) and by Netherlands Organisation for Scientific Research through project "Intersect". This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

REFERENCES

[1] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *OLS*.
[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *SOSP*.
[3] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. 2015. CAIN: Silently Breaking ASLR in the Cloud. In *WOOT*.
[4] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *IEEE S&P*.
[5] Sarah Cooley, Heidi Lohr, Peter Kral, Thomas Lee, and Benjamin Armstrong. 2018. Introduction to Hyper-V on Windows 10. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about.
[6] Will Gries, Jason Gerend, Cheryl Jenkins, Joseph Molnar, Mike Jacobs, and Elizabeth Ross. 2017. Data Deduplication Overview. https://docs.microsoft.com/en-us/windows-server/storage/data-deduplication/overview.
[7] Daniel Gruss, David Bidner, and Stefan Mangard. 2015. Practical Memory Deduplication Attacks In Sandboxed Javascript. In *ESORICS*.
[8] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*.
[9] David Kohlbrenner and Hovav Shacham. 2016. Trusted Browsers for Uncertain Times. In *USENIX Security*.
[10] Jens Lindemann and Mathias Fischer. 2018. A Memory-Deduplication Side-Channel Attack to Detect Applications in Co-Resident Virtual Machines. In *SAC*.
[11] Microsoft. 2017. July 18, 2017—KB4025334 (OS Build 14393.1532). https://support.microsoft.com/en-us/help/4025334/windows-10-update-kb4025334.
[12] Microsoft. 2018. PROCESS_MITIGATION_SIDE_CHANNEL_ISOLATION_POLICY structure (winnt.h). https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-process_mitigation_side_channel_isolation_policy.
[13] Microsoft. 2020. Deep Dive into Site Isolation (Part 1). https://microsoftedge.github.io/edgevr/posts/deep-dive-into-site-isolation-part-1.
[14] Microsoft. 2020. Memory Protection Constants. https://docs.microsoft.com/en-us/windows/win32/memory/memory-protection-constants.
[15] Grzegorz Miłós, Derek G. Murray, Steven Hand, and Michael A. Fetterman. 2009. Satori: Enlightened Page Sharing. In *ATC*.
[16] Mozilla. 2021. Cross-Origin Embedder Policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Embedder-Policy.
[17] Mozilla. 2021. Cross-Origin Opener Policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy.
[18] Mozilla. 2021. Firefox v94.0 Release Notes. https://www.mozilla.org/en-US/firefox/94.0/releasenotes.
[19] Mozilla. 2021. Multiprocess Firefox. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Multiprocess_Firefox.
[20] Mozilla. 2021. performance.now(). https://developer.mozilla.org/en-US/docs/Web/API/Performance/now.
[21] Mozilla. 2021. SharedArrayBuffer. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer.
[22] Marco Oliverio, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2017. Secure Page Fusion with VUsion. In *SOSP*.
[23] Rodney Owens and Weichao Wang. 2011. Non-interactive OS Fingerprinting through Memory De-duplication Technique in Virtual Machines. In *IPCCC*.
[24] The Chromium Project. 2020. Site Isolation. https://www.chromium.org/Home/chromium-security/site-isolation.
[25] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security*.
[26] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC*.
[27] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. 2021. Remote Memory-Deduplication Attacks. arXiv:2111.08553 [cs.CR]
[28] Prateek Sharma and Purushottam Kulkarni. 2012. Singleton: System-wide Page Deduplication in Virtual Environments. In *HPDC*.
[29] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. 2011. Memory Deduplication as a Threat to the Guest OS. In *EuroSec*.
[30] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. 2011. Software Side Channel Attack on Memory Deduplication. In *SOSP*.
[31] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. 2013. Implementation of a Memory Disclosure Attack on Memory Deduplication of Virtual Machines. In *IEICE FOECC*.
[32] VMWare. 2018. Security considerations and disallowing inter-Virtual Machine Transparent Page Sharing. https://kb.vmware.com/s/article/2080735.
[33] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. 2012. A Covert Channel Construction in a Virtualized Environment. In *CCS*.
[34] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. 2013. Security Implications of Memory Deduplication in a Virtualized Environment. In *DSN*.
[35] Jisoo Yang and Kang G. Shin. 2008. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-Page Basis. In *VEE*.