Timing-Based Browsing Privacy Vulnerabilities Via Site Isolation

Zihao Jin[†‡], Ziqiao Kong[†], Shuo Chen[†], Haixin Duan[‡]

[†]Microsoft Research Asia, Beijing, China [‡]Tsinghua University, Beijing, China

jinzihao1996@gmail.com, ziqiaokong@outlook.com, shuochen@microsoft.com, duanhx@tsinghua.edu.cn

Abstract – Chromium's site isolation ensures that different sites are rendered by different processes, which is a vision that academic researchers set forth over a decade ago. The journey from academic prototypes to the commercial availability represents a holistic rethinking about the security architecture for modern browsers. In this paper, we emphasize that the timing issues under site isolation need a thorough study. Specifically, we show that site isolation enables a realistic timing attack, which allows the attacker to identify which websites in a given target-sites set are loaded into the browser, as well as the website the user is currently interacting with. Through these vulnerabilities, the user's site-visit behavior is leaked to the attacker. Our evaluation using Alexa Top 3000 websites gives very high vulnerability percentages – 99%, 99% and 95% for our three key metrics of vulnerabilities. Moreover, the attack is very robust without any special assumption, so will be effective if deployed in the field. The main challenge revealed by our work is the tension between the scarcity of processes and the obligation to isolate cross-site frames in different processes. We are working with the Google Chrome team and Microsoft Edge team to propose and evaluate mitigation options.

1 Introduction

Site isolation is a major security feature that the Chromium team developed over the last several years. Since 2018, this feature is turned on by default in the Chrome browser. As Microsoft's Edge browser uses Chromium's rendering engine, the site isolation feature is also in effect in Edge. Mozilla's Project Fission is an effort to integrate site isolation into Firefox. The feature started to be officially tested in 2019 [5]. Site isolation has clearly become an industry-wide effort for browser security moving forward.

The essence of site isolation is to place contents from different sites in different OS processes, so that cross-site accesses must go through the process boundaries. The literature shows the benefit of having this process-level isolation when mitigating many types of vulnerabilities and attacks [19]. Site isolation is viewed as a fundamental technology offering important security values, so the Chromium team spent years to overcome significant performance and compatibility hurdles and achieve the goal.

Despite the exciting achievements of the technology, we show in this paper that many timing channels related to site isolation can be combined into a very reliable attack that spies on a user's browsing activity. Specifically, once a user visits the attacker's webpage or a third-party webpage containing the attacker's script, the attacker can identify which websites in a given target-sites set are being visited by the user's browser. The target-sites set may contain hundreds or a few thousands of sites that the attacker wants to monitor. Once the sites in the browser are identified, the attacker can monitor at real time which site is in the foreground tab, i.e., the one the user is interacting with. This allows the attacker to monitor how the user spends time between the sites on the monitored list. Compared to browsing privacy leaks in the past, this new attack is more powerful as the user discloses more detailed behaviors.

Overview of our work. In this study, we focus on the privacy implication of site isolation in light of cross-site timing. Our major effort is spent in understanding Chromium's cross-site communication mechanisms. Guided by the analysis of the IPC layer functions, we delve into many behaviors to examine if there are timing characteristics reliably observable by the attacker. We discover, for example, that the existence of a renderer process, the process priority of a renderer, and the "freshness" of a renderer are observable through cross-site timing measurements. These observations are used as the building blocks for our attack.

Although timing attacks are a familiar topic in the literature, many of them need strong assumptions about the victim's behavior and environment, thus tend to be fragile in practice (due to, for example, noise, jitters and workloads on the victim browser). The severity of our attack is demonstrated by its robustness. Our timing measurement techniques and the attack algorithm effectively avoid or mitigate the measurement fluctuations, and rely on multiple timing characteristics to cross-examine a situation. The algorithm, together with the insights about Chromium's internal, constitute the core technical value of our work.

Results. We have confirmed that the attack works reliably on Chromium/Chrome and Microsoft Edge with the default settings. The attack is fully automatic, and no prior knowledge about the victim is needed before the attack script lands onto the victim browser. Moreover, we do not assume that the browser contains only the attacker-site tab and the victim-site tab. The attack is robust enough so that workloads in other tabs do not cause sufficient impact on the effectiveness. This makes the threat very realistic.

The main results of our work are highlighted here:

(1) For the Alexa top 3000 sites, we test if the attack can confidently detect whether each site is in the foreground tab (FG), in a background tab (BG), or not present (NP). We show very high vulnerability percentages: over 99%, 99% and 95% of the sites allow the attack to differentiate confidently between BG vs. NP, FG vs. NP and FG vs. BG, respectively. A video demo about the attack scenario is available in reference [30].

(2) To demonstrate that the attack can simultaneously check a large set of target sites as a batch, we measure

the accuracy of the attack within roughly one minute when monitoring the Alexa top 500 sites altogether for their presence. Assuming the browser loads 5 out of the 500 sites, the chances of having no false positive and no false negative are 95.19% and 82.69%.

(3) To experimentally confirm site isolation as the cause of the problem, we test the attack against a wide range of Chromium versions. The result pinpoints to the earliest vulnerable version 68.0.3403.0, which is the version introducing site isolation [26]. We also confirmed that Edge and the experimental nightly build of Firefox are vulnerable.

(4) We are in a joint effort with the Microsoft Edge team and the Google Chrome team. The details of this work, including a full version of this paper, were disclosed to the two teams in early July 2020. The attack was confirmed. The Chrome team considers taking one of our mitigation suggestions. We are helping the team evaluate other mitigation options.

Paper organization. We give the background about site isolation in Section 2. Section 3 gives an overview about our attack. Many deeper insights are explained in Section 4, in which we delve into Chromium's internal to understand the observable timing characteristics. The evaluation results are shown in Section 5. Mitigation possibilities and our responsible disclosure to Google and Microsoft are presented in Section 6. Sections 7 and 8 give related work and conclusions.

2 Background about site isolation

Although the commercial availability of site isolation is very recent, the pursuit of its objectives has been over a decade. This journey represents perhaps the deepest holistic rethinking about the security architecture for modern browsers.

Browser used to be a single-process program. As the browser code became enormously complex, security bugs were inevitable, such as memory bugs (e.g., buffer overflows), same-origin-policy bugs (SOP bugs, e.g., universal cross-site scripting) and plugins' over-privilege issues. If an attack website could exploit a bug, the entire browser was in jeopardy. In response, security researchers began to experiment with multi-process architectures. OP [8] and Gazelle [23] browsers were two early prototypes treating different websites as distrusting principals at the OS level, i.e., isolating them in different processes. This is the same goal as site isolation, although achieving it in a commercial browser is a long journey.

Chromium before 2018. Chromium was the first commercial browser to adopt a multi-process architecture. However, it did not isolate websites, as the primary goal was for reliability. The Chromium team explicitly listed "origin isolation"[1] as an "out-of-scope goal" [3]. The entire web was treated as one OS principal, so an attack who compromised the rendering engine could access all the web contents inside the browser. The multi-process architecture did have an "in-scope goal" to separate the web from the local machine. However, researchers found that, without achieving site isolation, the web/local separation became problematic with modern cloud services integrated into local machines [11].

Chromium since 2018. With years of efforts, site isolation was made practical for Chromium. It was a major milestone of the decade-long pursuit for browser's solid security foundation. The Chromium team conducted a security evaluation [19] to show that site isolation could provide a range of mitigations, such as those for renderer vulnerabilities and transient execution attacks (e.g., Meltdown and Spectre [13][17]). With the evaluation result, site isolation clearly made the security benefit of the multi-process architecture compelling.

It is also worth noting that researchers considered site isolation as a countermeasure for timing attacks against pre-2018 versions of Chromium. For example, Vila and Köpf discovered in 2017 that Chromium was susceptible to a timing attack through the shared event loop in a process rendering contents from different sites. The attack could be mitigated by site isolation [22]. What our attack shows is that, although site isolation can be a countermeasure for Vila and Köpf's timing attack, it introduces a more dangerous and robust timing issue.

Chromium's process models and the new challenge. The Chromium team defines process models about how the web contents are partitioned in its multi-process architecture. The models include Single-process, Process-per-tab, Process-per-site and Process-per-site-instance. The concepts were first proposed in a research paper [18] and later described in a Chromium project documentation [24]. The default model is Process-per-site-instance, which creates a process for a set of connected pages from the same site. The set is called a *site instance*. Two pages are considered connected if they can reference each other using script code.

The process models were defined before site isolation was implemented. Site isolation brings a challenge to the process models because the intended granularity becomes frame/iframe, rather than tab. Process is a scarce resource in the OS. How should Process-per-site-instance be realized under the new granularity? For example, a typical CNN news page contains 15 iframes from 15 different sites (as a reference, this number for Fox News is 12). If a user opened 10 CNN pages, i.e., 10 site instances of CNN, then (15 + 1) * 10 = 160 processes would be created. It would be a 16-time increase of the number of processes compared to the previous Chromium versions, not practical for a general-purpose browser so widely used.

The tension between the scarcity of processes and the goal of isolating cross-site frames implies that some frames from the same site must share a process, even if they are in different site instances. This tension is an inherent challenge for the site isolation technology, and the root cause of the privacy issue we will discuss in this paper.

[1] The difference between "site" and "origin" is not essential in this paper. A site is a collection of origins sharing the same registerable domain and protocol. For example, a.google.com and b.google.com are two origins belong to the same site google.com.

3 Overview of the Attack

In this section, we give an overview of the attack, including the attack model, the basic operation, and the attack strategy. These understandings will set up the context for our investigation in Section 4 about Chromium's design and code.

3.1 Attack model and goal

The threat model of our attack is commonly known as the *web attacker model* in the literature [1], which assumes that the victim's browser, capable of script execution, visits the attacker's website http(s)://attacker.com, or a website containing a script of the attacker. The attacker is unable to run any binary code in the victim browser, or eavesdrop/intercept the network communication.

Figure 1 shows the attack scenario: there is an *attacker tab* running the attack script. The attacker's goal is to monitor a set of sensitive target sites – victim1.com, victim2.com, ..., and victim500.com. In Figure 1, the user is visiting victim1.com and victim2.com. We call their tabs the *victim tabs*. The victim tabs and the attacker tab can be in the same browser window, or different browser windows belonging to the same browsing instance (note that an "incognito" window runs as a separate browsing instance). Moreover, a victim site does not need to be loaded into the main frame of the tab.

In the attacker tab, a script iteratively uses an invisible iframe to visit a list of URLs (one at a time), and measures the times for loading completion. The focus of this section is to explain how the script strategically chooses the list of URLs to take the measurements, so that the combinations of the measurement results can robustly achieve the attack goal, despite the big variety of network environments and local machine/workload conditions.

For the scenario in Figure 1, the goal of the attack script is: (1) to determine that victim1.com and victim2.com, but not other target sites, are present in the browser instance; (2) to monitor and determine which site is in the foreground tab. For example, victim1.com currently is. This allows the attacker to observe the user's behavior effectively.

To demonstrate the severity of the privacy problem, our objective is to show that the attack can be launched at a large scale. Therefore, our attack model requires the script to be fully automatic, and has no input parameter. This makes our result convincing – no prior knowledge about a victim's circumstance is needed before the script lands on the browser.

3.2 Basic operation – the loading-time measurement

We mention the process model in Section 2, as well as the challenge due to the tension between site isolation and the process scarcity. A question arises – whether site isolation introduces an exploitable timing channel. The answer is yes. The timing channel can be easily confirmed by measuring the time for loading a URL of victim site into the iframe on the attacker tab. Specifically for the example in Figure 1, when the script measures the time for the iframe to load a victim1.com URL versus a victim500.com URL (note that victim500.com is not being visited by the user in this example), the time

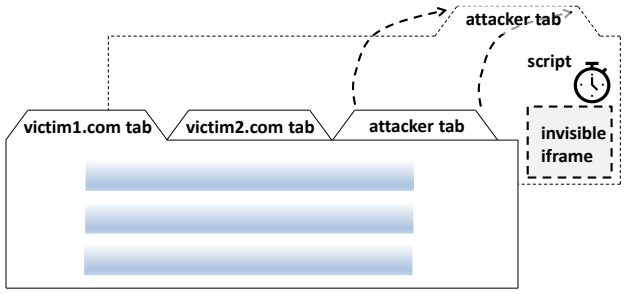


Figure 1: The attack scenario

difference is noticeable. This is because the process for processing victim1.com already exists, but a new process needs to be created in order to process victim500.com.

Timing channels generally exist to various extents in multi-process systems, so its existence with the site isolation mechanism may not be immediately concerning. The important question is whether a robust and fully automatic attack can be built. The rest of this section and Section 4 explain how such an attack is fulfilled with insights about the internals of Chromium.

The pseudo code in Listing 1 shows the basic operation: measuring the "relative time" r, defined later, for loading a URL from victim_site.com (denoted as vtm). The calculation requires another domain reference_site_1.com (ref_1) that we register in advance. The attack page first includes an empty page from ref_1, then repeatedly creates two iframes navigating to ref_1 and vtm, and measures their loading times in their onload event handlers, then removes the iframes. Note that the two iframes are created sequentially – the second iframe is created only after the first one fires its load event. This avoids the interference between the two measurements. The explanation of this code is given next.

Network jitter. In a real-world scenario, the network jitter can easily add hundreds of milliseconds, seriously affecting the timing accuracy. There are techniques to avoid the network jitter. For example, we use the "unsafe port" technique as follows. Chromium defines a list of unsafe ports for HTTP(S) requests, including port 1, 22, 23, 25, etc. Requesting a URL with an unsafe port would be early terminated – it still initializes a renderer process as if it is navigating to an ordinary page, but does not make actual network requests. In this way, we can reliably measure renderer initialization time by completely avoiding the effect of network jitter. In the rest of this paper, when we use the phrase "loading a URL", it means "loading a URL with an unsafe port", and the phrase "loading a page" means "loading a valid page". In Section 5.2, we will show that the attack does not fundamentally depend on using an unsafe port. The attacker can utilize the browser's disk cache to avoid the network jitter effectively. The attacker script can load a static resource in the victim site (e.g., *favicon.ico* or *robots.txt*) for the first time. All subsequent loadings of this resource will hit the disk cache without causing any network request.

Local machine fluctuation. The local machine fluctuation, either due to unrelated sites' activities, or other programs running on host OS, can interfere with the timing. To minimize

the impact, we repeat the measurement for 10 times, take the median value of each frame's loading time as final result. Then we calculate a *relative time* for loading a vtm URL, using Equation 1 with pseudo code in Listing 1.

$$r_{vtm} = \frac{\text{loadingTime}(vtm)}{\text{loadingTime}(ref_1)}$$

Equation 1: Calculation of the relative time for loading *vtm*

```
INIT:   <iframe src="http://reference_site_1.com/empty.html">
REPEAT:
    time_0 = performance.now();
    <iframe src="http://reference_site_1.com:1"
      onload="time_ref = performance.now()">
    <iframe src="http://victim_site.com:1" onload="time_vtm =
      performance.now()">
    /* Note that the two iframes are loaded sequentially, so
      Equation 1 is calculated below. */
    r = (time_vtm – time_ref) / (time_ref – time_0)
```

Listing 1: iframe loading time measurement

When multiple sites are loaded together, they are both slowed down if there is a temporary surge in CPU usage. In this situation, the victim site's absolute loading time can increase significantly, but the relative loading time is much more stable.

3.3 The attack algorithm

Baseline. The goal of the baseline measurements is to obtain r_{np} and r_{bg}. The former, in which "np" stands for "not-present", is the relative time for loading a URL of a site when the renderer process for the site has not been created yet. In other words, no other tab has loaded any page from this site yet. The latter is the relative time for loading a URL of a site when the renderer process for the site has been created to render an existing page that is currently invisible ("bg" for "background").

The calculation of r_{np} is done using Equation 1, in which vtm is replaced by another domain reference_site_2.com (ref_2) that we register. Because ref_2 is a domain that no user knows about, it is valid to assume that it has not be present in the browser before our attack script loads it.

The algorithm can now calculate r_{bg}. To do it, the script creates another invisible iframe to load ref_2 again. Same as the measurement for r_{np}, Equation 1 is used with vtm replaced by ref_2. The measurement result is r_{bg}. The value is different from r_{np}, because the renderer process for ref_2 has already existed prior to this measurement, i.e., r_{bg} does not include the time for renderer process creation. Moreover, since both measurements are done using invisible iframes, no time is consumed due to user interface (UI) functionalities. We will explain that the UI time allows us to differentiate foreground and background tabs.

After obtaining r_{np} and r_{bg}, we store their values in cookies for later measurements. Note that the relative loading time (r) differs across different machines and operating systems, but it is stable across different measurement runs on the same installation of Chromium. Therefore, for each victim browser, the baseline measurements only need to be performed once.

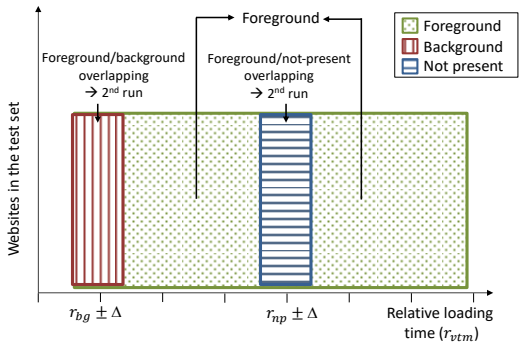


Figure 2: Conjectured distribution of r_{vtm} from first run

First run. With the baseline results in the cookies, the attack starts. It uses the invisible iframe (as shown in Figure 1) to load a URL of vtm. Equation 1 gives the value of r_{vtm}. Figure 2 shows our conjectured distribution of r_{vtm}. The X-axis is for r_{vtm}, and the Y-axis represents the websites in the test set. If the victim site is not present, r_{vtm} should be close to r_{np}. In the figure, the left stripe indicates the range (r_{bg} ± threshold), in which threshold is anticipated to be very small. We conjecture that, given a site vtm in the test set, if there is already a background tab containing a page from vtm, r_{vtm} should usually fall into the left stripe. Similarly, if no tab contains a page from vtm, r_{vtm} should usually fall into the right stripe (r_{np} ± threshold), which is anticipated not to overlap with the left stripe. However, if it is the foreground tab that currently renders a page from vtm, we do not know where the measured relative time will fall in the figure, because the time due to UI functionalities varies significantly depending on the page's complexity. Therefore, we use the shaded area to indicate the range of r_{vtm} in this situation, which overlaps with the two stripes. If a measurement does not fall into the two stripes, it indicates that the foreground tab is rendering a page from the site vtm; if it does, the attack script needs to use the second measurement, described next, to differentiate the situations.

Second run. Two cases need the second run.

Case 1: If r_{vtm} falls into the left stripe, it indicates that a page from vtm has already been loaded into one of the tabs, but it is not clear whether the tab is in the foreground or not. We will explain in Section 4.4.3 that Chromium assigns a higher process priority for the foreground renderer, which helps the foreground renderer to maintain its responsiveness when the CPU is under a heavy pressure. Our attack exploits this timing channel to differentiate the "foreground" and "background" situations. It is accomplished as follows. To generate CPU pressure, we use an iframe to include a helper page from another domain (so that it runs in a renderer process different from the attack page) which creates multiple JavaScript workers running in an infinite loop to give the CPU a heavy pressure. The content of the helper page is unimportant. Under the CPU pressure, a foreground renderer's responsiveness is degraded much less as compared to a background renderer, thanks to Chromium's process priority strategy (additional details in section 4.4.4). Referring to Equation 1, since ref_1 is known to be in the

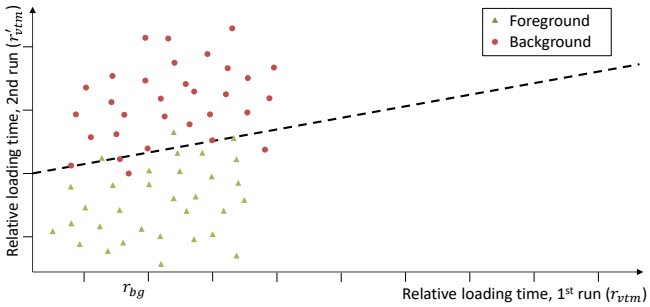


Figure 3: Conjectured distribution of r_{vtm} and r'_{vtm}

background, if the measurement result, denoted as r'_{vtm}, does not decrease noticeably, it indicates that vtm is also in the background. Otherwise, it is in the foreground.

Figure 3 shows the conjectured distribution of the measurements in the space using the values of the first and second runs as the two dimensions. The effectiveness of the attack depends on how well the space can be partitioned into two regions, so that most BG dots are in one region, and most FG dots are in the other. As we will show in Section 5, a straightforward binary classification using SVM (Support Vector Machine) is already effective. It simply partitions the space using a straight line. We will report the precision and recall rates in Section 5.

Case 2: If r_{vtm} of the first run falls into the right stripe in Figure 2, we need to determine whether this is a "foreground" or "not-present" situation. The timing channel that our attack exploits is the renderer's HTML parsing performance. We will show in Section 4.4.5 that, in the period shortly after a renderer is created, its HTML parsing performance is considerably lower than that of a renderer already running for a while. This enables a robust measurement to differentiate between the "foreground" and "not-present" situations. The measurement, denoted as r''_{vtm} in Equation 2, uses three iframes. The first of them loads a URL of ref_1, same as the basic operation described in Section 3.2. After ref_1 finishes loading, the other two iframes are added to the page at once, with each loading a URL of vtm. The loading times of the two vtm iframes are denoted as loadingTime(vtm) and loadingTime'(vtm). Deferring many technical details to Section 4.4.5, we will explain that the denominator loadingTime(vtm) measures the time of the entire iframe loading, while the numerator (loadingTime'(vtm) - loadingTime(vtm)) measures the HTML parsing time, which is the final stage of the iframe loading.

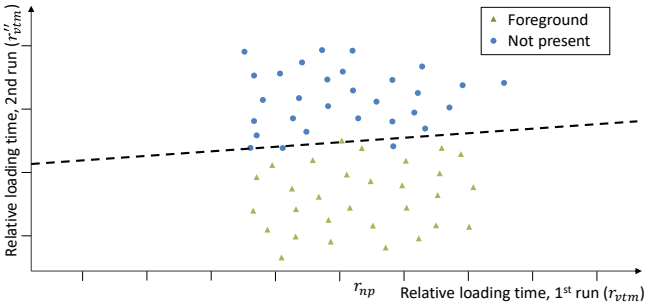


Figure 4: Conjectured distribution of r_{vtm} and r''_{vtm}

$$r''_{vtm} = \frac{\text{loadingTime}'(vtm) - \text{loadingTime}(vtm)}{\text{loadingTime}(vtm)}$$

Equation 2: Relative time for the HTML parsing

A higher r''_{vtm} indicates a longer HTML parsing time, i.e., a lower HTML parsing performance, which indicates that the renderer is newly created (most likely due to our measurements). Hence it is classified as a "not-present" situation. On the other hand, a lower r''_{vtm} indicates the renderer has existed for a while prior to our measurements. Since we are in case 2, this means the "foreground" situation. Similar to case 1, a conjectured distribution for case 2 is shown in Figure 4, in which the Y-axis represents r''_{vtm}.

Pseudo code. The pseudo code in Listing 2 summarizes the attack algorithm, including baseline, the first run, and the two cases in the second run. The function calculate_SVM_params calculates the separating hyperplane (i.e., the dashed lines) in Figure 3 and Figure 4. Each result is defined by two parameters A and B. Because the SVM classifier is pretrained, this function only needs to do a quick adaption with the victim's baseline values r_np and r_bg to target the victim browser. The value #CPU is the number of CPU cores on the victim's computer. The API navigator.hardwareConcurrency provides the value. Empirically, we find creating one fewer JavaScript workers than CPU cores can apply an adequate CPU pressure to the victim renderer, without making the attack page unresponsive.

Multiple windows. It is worth noting that this section describes the one-window scenario, for the purpose of simplicity. If the victim user opens multiple windows, it is possible that the victim site is rendered by multiple foreground tabs, which can (but not necessarily) be handled by different

```
const: DELTA
r_np, r_bg = load_baseline_results()
if r_np is None or r_bg is None {
   r_np = equation_one("reference_site_2.com")
   create_invisible_iframe("reference_site_2.com")
   r_bg = equation_one("reference_site_2.com")
   save_baseline_results(r_np, r_bg)
}
a_1, b_1, a_2, b_2 = calculate_SVM_params(r_np, r_bg)
r_vtm = equation_one("victim_site.com")
if abs(r_vtm – r_bg) < DELTA {
   create_invisible_iframe("helper_site.com", workers = #CPU-1)
   r_vtm_1 = equation_one("victim_site.com")
   if a_1 * r_vtm + b_1 * r_vtm_1 < 1 :
      output("foreground")
   else:
      output("background")
} elif abs(r_vtm – r_np) < DELTA {
   r_vtm_2 = equation_two("victim_site.com")
   if a_2 * r_vtm + b_2 * r_vtm_2 < 1 :
      output("foreground")
   else:
      output("not present")
} else:
   output("foreground")
```

Listing 2: The attack algorithm in pseudo code

renderers. When this situation happens, it will be observable as r_{vtm} (i.e., the first run value) jumping between multiple values, with each value reflecting the page complexity of one foreground tab. We will further explain this in Section 4.4.2.

3.4 Batch testing for presence of target sites

The above steps describe how to monitor a single site. In fact, the first run can be done in a batch manner for a set of sites (in our experiment, we do it for 500 sites within one minute). This allows the attacker to know which sites in a large set of target sites are present. Admittedly, the site in the foreground tab has a small possibility to be detected as not present, if it happens to have the FG-NP overlapping. However, because there is only one foreground tab per window, misidentifying an FG case as an NP case has a very small impact on the overall accuracy of the batch test. Moreover, if the attacker really wants to minimize the chance of misclassification, he can build a list prior to the attack, which contains all the target sites having the FG-NP overlapping. These sites need the "second run, case 2" test after the batch test.

The purpose of the batch test is to quickly narrow down to the present sites, so that the attack script can monitor them closely with the more expensive "second run" techniques.

There are technical details to make the batch as fast and accurate as possible. They are summarized below:

(1) It is feasible to measure two sites in parallel without making the browser less responsive. Responsiveness is a requirement for the accuracy of the measurements. Of course, maintaining the responsiveness also makes the attack stealthy – a normal user is unlikely to notice its existence.

(2) The basic operation described in Section 3.2 is used as a measurement run, which gives the r_{vtm} value. Given the time budget, our strategy is to spend more measurement runs on sites that are more likely present. Suppose the set contains 500 target sites. The algorithm is: all 500 sites are measured once; among which the 250 sites with lower r_{vtm} values are measured again, among which 125 sites with lower r_{vtm} values are measured again, and so on, until the number of sites gets to 1. With this strategy, the average number of measurements for a single site is 2, but low r_{vtm} sites (i.e., the likely-present sites) are measured many times. We find that it results in a good accuracy within the small time budget.

4 Deeper into Chromium's Design and Code

Both the algorithm and the detailed steps described in Section 3 are the result of our deep investigations about cross-process timing characteristics of Chromium, which is the focus of this section. It is worth emphasizing that timing differences are not hard to find when we broadly investigate browser behaviors, but identifying the robustly exploitable ones is the key in our research.

4.1 Investigation approach and tools

The codebase of Chromium has over 25 million lines of code. To study the timing issues introduced by site isolation, we start from understanding the major components in the Chromium architecture and how they interact. It requires a great amount of effort to read standards and developers' forums/ documentations, and to step through the actual code. The effort enables us to locate the focused areas related to cross-site communication, which is described in Section 4.2 and Section 4.3. The detailed investigations are described in Section 4.4, which delves into the Chromium's internal to explain why the attack in Section 3.3 is effective and robust.

Besides reading documentations and code, we use and develop tools for the investigation. First, we use LibClang to parse Chromium's source code, and create an index of tokens and literals, which are mapped to their locations in source files. When we try to locate the code responsible for a behavior of Chromium, a keyword search in this index gives us a short list of possible locations in source code. For those functions and variables that we suspect to leak sensitive information, we first recursively identify where it is referenced using this index, skip those with a limited scope that does not involve any sensitive information. Then, we add instrumentation code to the function under investigation to get the stack trace when it is invoked. After building and running the instrumented version, we convert memory addresses in the stack trace to locations in source files, utilizing the DWARF debug information compiled into Chromium's binary. This enables us to map the attacker invoked behaviors to their underlying code paths.

4.2 Cross-site communication

The essence of the attack is that the attacker measures timing characteristics of the behaviors of the victim-site's renderer process. It is important to identify the mechanisms for cross-site communications. Given Chromium's huge codebase, the investigation could easily become aimless. What we realize is that IPC is a highly desirable layer for our investigation, because: (1) since sites are isolated by processes, all cross-site communications need to be translated into IPC calls; (2) Chromium currently has two IPC modules – legacy IPC and the *mojo* IPC, and is gradually migrating from the former to the latter. The fact that IPC can migrate suggests that the IPC interface is relatively clean, allowing us to better understand and runtime-trace the IPC calls; (3) although the number of IPC message types is big, it is still feasible to enumerate them. Therefore, it is feasible to achieve completeness if researchers continue in the direction we have been taking.

Specifically, we insert a hook to the function used to create mojo IPC connections, and capture 38 mojo receivers and 53 mojo remotes bound by any of the renderers during a browsing session. (A mojo receiver is a group of functions callable by a remote process, while a mojo remote is a proxy to a group of functions implemented by a remote receiver.) Among them, we identify a mojo receiver and a remote that carry all legacy IPC messages, as a part of the transition from the legacy IPC to the mojo IPC. We find no direct IPC connections between renderers, which means all communications between renderers go through browser process. With static analysis, we identify 768 legacy IPC message types, each representing an action that can be invoked, or a type of data that can be sent between processes.

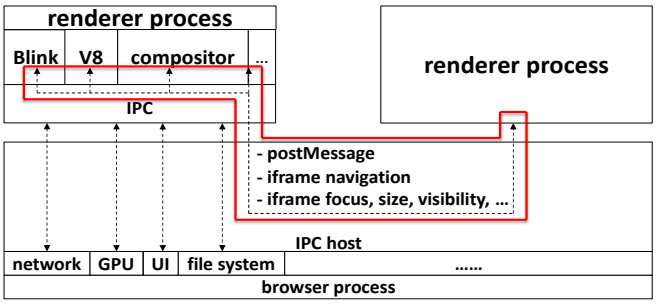


Figure 5: focused area of our investigation

Focusing on IPC helps us narrow the investigation space. We study every cross-site communication with a corresponding IPC interface pair, one from renderer to browser, and the other from browser to renderer. Figure 5 highlights this space in Chromium's architecture, which is obviously a great reduction from Chromium's 25 million lines of code. In the following subsections, we present our findings in this space.

4.3 PostMessage

As postMessage is one of the few legitimate ways of cross-origin communication defined by HTML standard [14], and cross-site interaction is a subset of cross-origin interaction, we use it as a good entry point to understand the implementation-level changes introduced by site isolation. Note that postMessage is not used in our attack described in section 3, but it guides our investigation and can be a building block for future cross-site attacks that target implementation-level weaknesses.

Defined by the HTML standard, postMessage is a method of the Window object. Internally, Chromium implements a DOMWindow class, with a member function postMessage. After site isolation is implemented, postMessage can be classified into 3 cases: (1) cross-site, (2) cross-origin but same-site, and (3) same-origin. Based on public documentations, we infer that, before site isolation, Chromium's postMessage implementation checks the message and the recipient origin as required by the HTML standard, but there is no distinction between case (1) and (2). At implementation level, this distinction is made by deriving two subclasses from DOMWindow, which are named LocalDOMWindow and RemoteDOMWindow. The former lives within a renderer and inherits the origin check mechanism from the base class DOMWindow. The latter is conceptually a proxy to a LocalDOMWindow that lives in another renderer, while the data is forwarded by the browser process, as there are no direct communication channels between renderers. The browser process checks a message's target site to deliver it to the right renderer, which checks the message's target origin before passing it to the onmessage event handler.

The understanding of postMessage guides our study about more complex cross-site communication mechanisms, which are presented in the following subsections. They all share similarities with postMessage: the browser process is only responsible for delivering the cross-site message to the right renderer, so that its handling logic is kept minimal. We find that the amount of new code added by site isolation is relatively small for handling a cross-site message. It is a thin wrapper layer calling into the existing code that predates site isolation. Therefore, our study of postMessage gives an important insight: the study about timing issues should not only focus on the thin layer of site isolation code, but needs to go deeper into the existing code to analyze detailed timing characteristics of the renderer's behaviors, some of which are described next.

4.4 Cross-site iframe navigation

Even before postMessage was introduced into browsers, embedding a web page from a different site had already been a mechanism for cross-site communication. Our attack shown in Section 3 uses this mechanism – the host page requests the embedded iframe to load and render a given URL, which is a complex process for modern browsers. Next, we show what happens inside a renderer when it loads an iframe, and what kinds of internal states might be leaked to the host page.

4.4.1 Renderer allocation

Cross-site iframe navigation begins with the host page's renderer sending a BeginNavigation IPC request to the browser (i.e., the browser process). After some validity checks, the browser updates its frame-process mapping, then dispatches the request to the embedded page's renderer, like in postMessage.

The browser is responsible for keeping track of a site's renderer processes. It first tries to assign the navigation task to a visible (i.e., on the foreground tab) renderer hosting that site. When there are multiple visible renderers (i.e., when the user opens multiple windows, there can be multiple foreground tabs), it randomly selects one. If it cannot find a visible renderer, it then tries to randomly select a hidden (i.e., on a background tab) renderer. If it fails again, then the browser launches a new renderer process to handle this navigation request.

Since launching a new process introduces a millisecond-level latency, it is observable by the host page from onload event handler. Tens of milliseconds (around 60ms on a modern Intel i7 CPU) saved on a cross-site iframe navigation indicates that the target site already has an instance, which is either a tab or an iframe. This enables a web attacker to detect any other site's existence on the user's browser, without requiring the consent from the user or the target site.

Note that the aforementioned renderer reuse policy only applies to iframe navigation. Navigation by clicking on a link, or directly entering a URL into address bar is subject to different policies. In these situations, Chromium can create a new renderer even if there is an existing same-site renderer. Therefore, site isolation does not mean a one-to-one mapping between sites and renderers, since Chromium does not always attempt to reuse an existing renderer for all types of navigations.

4.4.2 RenderView initialization

Either by launching a new renderer or reusing an existing one, the browser can always find a renderer to handle the iframe navigation. Then, it passes on all the necessary information to the renderer through CreateView IPC interface. (Note that if the host page already has an iframe of embedded site, creating a second iframe of that site will not trigger CreateView.)

```
static void blink::SetSelectionColors() {
  for (Page* page : AllPages()) {
    for (Frame* frame = page->MainFrame(); frame != NULL; frame
= frame->Tree().TraverseNext()) {
      /* update color setting here */
      if (typeof(frame) == typeof(LocalFrame)) {
        frame->GetDocument()->PlatformColorsChanged();
}}}}
```

Listing 3: global settings update in CreateView

When we investigate the process of creating a RenderView, we are particularly interested in whether the process involves existing frames in the renderer. If this is the case, it may reveal information of other tabs or iframes of that site. We find that, inside CreateView, there is a call to blink::SetSelectionColors, which is a static function of Blink (Chromium's rendering engine) that updates a global setting shared by all frames in the renderer. Listing 3 shows a pseudo code snippet based on Chromium's source code. As the code shows, after it updates global color setting, it notifies every frame through its PlatformColorsChanged method. In this way, it affects all frames in a renderer, not only the newly-created frame.

Note that every frame has updated its selection colors, the renderer needs to recalculate the style for each element in each frame. However, to avoid unnecessary computation, it would only redraw visible frames. The time to redraw a visible frame depends on its visual complexity. Complex pages typically found on video sites (e.g., YouTube) can consume additional hundreds of milliseconds because of this when it is visible.

This behavior has two implications. First, a web attacker can detect if a victim site is on the user's foreground tab, as long as the page is reasonably complex. Second, it also reveals the visual complexity of a victim site's page, which suggests the type of content the user is currently viewing.

Multiple frames. The victim site may have multiple frames that belong to different renderers. In the simple case, when it has only one visible frame, as described in Section 4.4.1, Chromium always selects the renderer hosting this visible frame for a new iframe navigation task. Therefore, the redrawing time of that visible frame will always be reflected when we create a new iframe of that site. However, when the user opens multiple windows, there can be multiple visible frames of victim site. If the visible frames belong to the same renderer, the redrawing time will be cumulative (i.e., reflecting the combined complexity of all visible frames) when we create a new iframe. Otherwise, if they belong to different renderers, as described in Section 4.4.1, Chromium will randomly select a renderer from them when a new iframe is created. In this situation, every time an iframe navigation occurs, it can indicate the redrawing time of visible frames in a different renderer.

4.4.3 Renderer state

A renderer process has two pairs of states: visible-hidden and foreground-background. These two pairs of states are shared by all frames handled by a renderer. A renderer is visible whenever one of its frames is visible, and a renderer is foreground if it is visible or playing sounds. When a renderer is in foreground state, its process priority is set to high. The high priority can speed up iframe loading, and the difference becomes more noticeable when the system load is high.

Note that a renderer's visibility can affect its iframe loading time in both directions, depending on the visual complexity of the contained page. When comparing the times for an invisible frame and a visible frame to load the same page, we can see that the former is noticeably faster when the page is complex (as discussed in Section 4.4.2), and the latter is noticeably faster when the page is simple. These two observations combined enable a web attacker to detect the visibility of another site with a high confidence.

4.4.4 Load event

When the embedded page of an iframe finishes loading, it notifies the host page with a load event. For cross-site iframes, this load event needs to cross process boundary to reach the host page's renderer. Similar to postMessage, the embedded page sends a DispatchLoad request to browser process, which will then dispatch the load event to the host page through another IPC interface.

In other words, the iframe navigation in the attack scenario is a two-way interaction between an attacker renderer and a victim renderer. As Figure 6 shows, by observing the time between BeginNavigation and DispatchLoad, the attacker can infer the characteristics of the sequence of actions taken by the victim renderer when navigating to an attacker-specified URL.

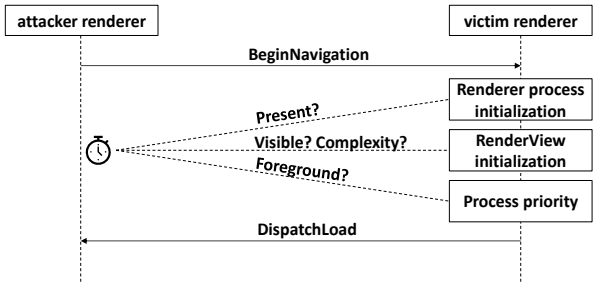


Figure 6: timing about iframe navigation

4.4.5 The HTML parsing performance

In previous subsections, we focus on identifying time-consuming operations in a "full" iframe navigation, including the initialization of renderer process and RenderView. As we investigate deep into the code execution, we see that the iframe loading time contains 5 stages. When multiple iframes of a site are loaded concurrently, they are pipelined by the renderer, so the stages of the loading tasks are interleaved.

Figure 7 shows a trace captured with Chromium's trace event profiling tool (screenshot of chrome://tracing). The texts in the screenshot are too small to be legible, so we annotate the key information above and below the screenshot. The trace is generated from concurrently loading two URLs of the same site into two iframes (referring to Section 3.2 for the meaning of "loading a URL"). Suppose the two iframes fire the two onload events at time1 and time2. It is easy to see that (time2 - time1) indicates this renderer's time consumed by stage 5 of the second

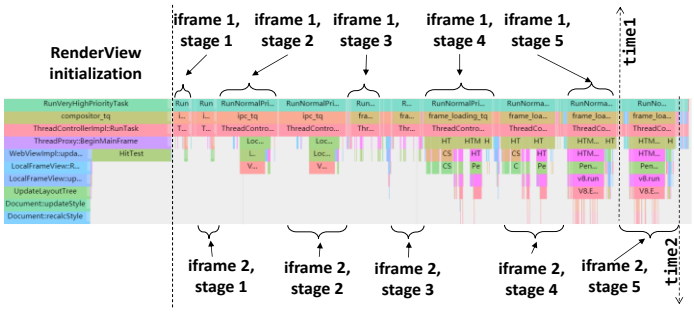


Figure 7: iframe loading pipeline (screenshot of chrome://tracing)

concurrent task. Inspecting the call stack of stage 5 (which is too small to read in Figure 7), we understand that the stage is for HTML parsing, which means that the attacker can measure the victim renderer's HTML parsing performance. As described in Section 3.3 (Case 2 of the Second Run), we rely on this feature to distinguish between a newly-created renderer and a long-existing one, as we observe that a newly-created renderer spends noticeably more time for the HTML parsing.

5 Evaluations

5.1 Evaluation using Alexa top 3000 sites

To quantitatively measure the effectiveness of the attack, we use Alexa top 3000 sites as our test set. We collect four datasets for the evaluation, as described below.

5.1.1 Experiment setup and datasets

The browser is Chrome 87 running on Windows 10. The datasets are collected on two machines. One is a *desktop* PC with the specification Intel i7 8700 (6 cores, 12 threads @ 3.2GHz), 16GB memory; the other is a *laptop* PC with the specification Intel i7 5600U (2 cores, 4 threads @ 2.6GHz), 8GB memory. To represent different background workloads, the datasets include the one-extra-tab and five-extra-tabs scenarios. In the latter, the browser contains the attacker's tab, the victim's tab, and five extra tabs loading five random sites from Alexa top 1,000,000 sites. The former contains only one such tab. The purpose is to confirm that our attack's accuracy is not affected by the number of extra tabs.

The evaluation is carried out using an automatic script. The process for the five-extra-tabs scenario is:

(1) Start Chromium, open five new tabs to load the homepages of five random sites from Alexa top 1,000,000 sites (tabs 1-5).
(2) Open a new tab for the homepage of a victim site (tab 6).
(3) Open another new tab, navigate to the attack page (tab 7).
(4) Switch to tab 1 to cause a BG situation. Measure r_{bg}.
(5) Switch to tab 6 to cause an FG situation. Measure r_{fg}.
(6) Close tab 6 to cause an NP situation. Measure r_{np}.
(7) Switch to tab 7, save output, exit Chromium. Go to (1).

The four datasets we collect are: (a) Desktop five-extra-tabs, (b) Desktop one-extra-tab, (c) Laptop one-extra-tab, (d) Laptop five-extra-tab. Datasets (a)(b)(c) are collected in January 2021. Dataset (d) are collected in July 2021.

5.1.2 Results

The results of the four datasets are similar. Figure 8 ~ Figure 11 show the results of dataset-(a). These figures can be viewed more clearly if color-printed. Figure 8 shows the distribution of relative loading time (r) in different situations: "foreground (FG)", "background (BG)" and "not-present (NP)". For every site foo.com, the script tries to visit *http(s)://foo.com*. There are 225 sites not returning valid pages for the URLs, because they do not exist or are temporarily inaccessible, so dataset-(a) contains 2775 valid sites.

As expected, the r_{np} and r_{bg} values of most sites fall into the two narrow stripes. There are 21 sites having r_{np} and r_{bg} values that we consider as outliers. Even though most of the outlier values are not reproducible upon re-measurements, we still count them all against our accuracy. Also, we conservatively assume that the accuracy for these websites is 0%, although a blind guess of NP or BG would yield an accuracy better than 0%. Bearing this accuracy penalty, we remove the outliers, and produce Figure 8 ~ Figure 11.

Background vs. not-present. Hiding the FG data points in Figure 8, we produce Figure 9 to show the clear distinction in the relative loading times for the BG and NP situations. The accuracy shown in the figure is 100%. With the penalty due to the outliers, the accuracy is (2775 – 21) / 2775 = 99.24%.

Foreground vs. background. The BG stripe is narrow. There are 890 FG points falling into the BG stripe, which need to be further analyzed by the "second run, case 1" method described in Section 3.3. The method increases the CPU pressure without causing unresponsiveness. As explained earlier, the foreground renderer has a high process priority, so the second-run measurement r' tends to be lower in an FG situation, as compared to a BG situation. Similar to the 21 outliers in the first run, we discard 10 outliers in the second run, and count them against us with a 0% accuracy. Figure 10 shows the hyperplane drawn by the SVM algorithm to separate the 880 BG points and the 880 FG points. The accuracy shown in the figure is 93.47%. In summary, the end-to-end FG-BG accuracy is (880 × 93.47% + (21 + 10) × 0% + (2775 – 880 – 21 – 10) × 100%) / 2775 = 96.81%.

Foreground vs. not-present. Although the NP stripe is wider than the BG stripe, there are only 43 FG points falling into the NP stripe. They are further analyzed using the "second run, case 2" method described in Section 3.3. It uses the HTML parsing time to indicate the "freshness" of the renderer. The method is highly effective. We do not need to remove any outlier. Figure 11 shows that the SVM algorithm draws a hyperplane, and all points are correctly separated. Therefore, the end-to-end FG-NP accuracy is (43 × 100% + 21 × 0% + (2775 – 43 – 21) × 100%) / 2775 = 99.24%.

Comparison of the four datasets. The figures for datasets (b), (c) and (d) are shown in the appendix. The accuracy numbers are shown below. Comparing dataset-(a) and dataset-(b), we can see that the impact of the pages in the extra tabs on the accuracy is too insignificant to be statistically meaningful. Dataset-(c) is collected from the laptop PC less powerful than the desktop PC for dataset-(a) and dataset-(b). The slight decrease of the FG-BG accuracy (e.g., 91.81% in Dataset-(d))

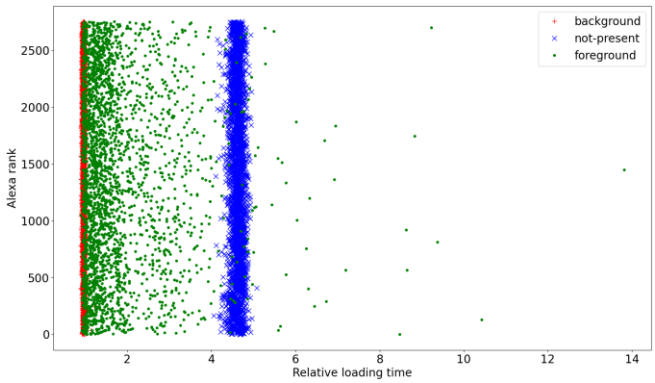


Figure 8: loading time distribution, dataset-(a)

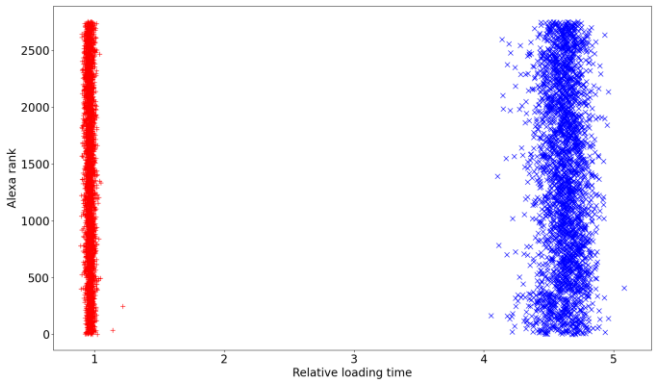


Figure 9: Background vs not-present, dataset-(a)

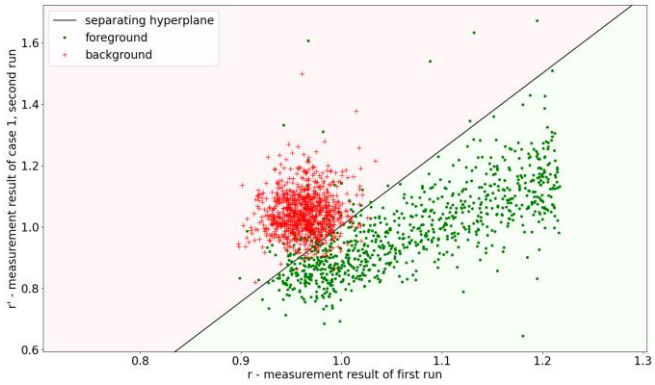


Figure 10: foreground vs background, dataset-(a)

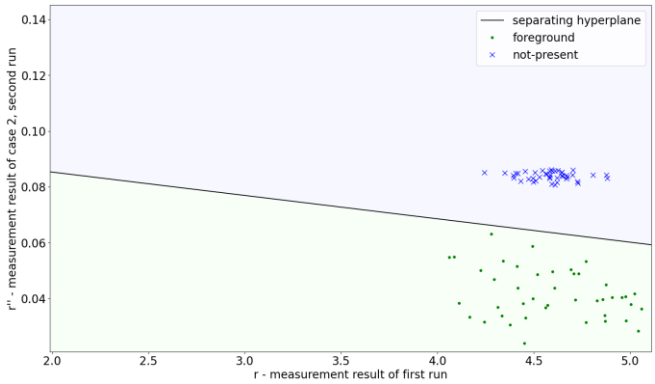


Figure 11: foreground vs not-present, dataset-(a)

may suggest that our current setting for the CPU stressing is slightly too heavy for the laptop, affecting the measurements.

	BG-NP	FG-BG	FG-NP
Dataset-(a)	99.24%	96.81%	99.24%
Dataset-(b)	99.36%	97.60%	99.36%
Dataset-(c)	99.24%	95.07%	98.72%
Dataset-(d)	99.22%	91.81%	98.99%
Average	99.26%	95.32%	99.08%

Other machines tested. In addition to the two desktop and laptop machines, we also use several other machines with a variety of specifications to do manual testing. We do not collect datasets for the statistical evaluation, but only validate that the same attack works on these machines. A subset of machines are described in Table 1. The tests were performed from locations in Asia and America. The environments included home, university and corporate networks, with and without VPN.

Surface Book: i7-8650U @ 1.90GHz, 16GB RAM
Surface Pro 4: i5-7300U @ 2.60GHz, 8GB RAM
Surface Pro 3: i5-4300U @ 1.90GHz, 8GB RAM
HP Pavilion 690-0024: AMD Ryzen5-2400G @ 3.60GHz, 8GB RAM
ThinkPad T490: i7-8565U @ 1.80GHz, 16GB RAM
A desktop machine: Intel Xeon E5-1620 v4 @ 3.50GHz, 32GB RAM

Table 1: A subset of machines used for manual testing

5.2 Using the disk-cache instead of the unsafe port

In Section 3.2, we mention that the basic operation does not fundamentally depend on the "unsafe port" behavior. Another technique to avoid the network jitter is to use the disk cache. The effect of the disk cache is that, if a resource is in the cache, a subsequent loading of the resource will be completed without having any network request, which serves the same purpose as the "unsafe port" behavior.

The requirements for the resource. The attack requires a resource that satisfies the following requirements:

(1) X-Frame-Options and Content-Security-Policy allow the resource to be framed inside a page in another site;

(2) The cache-control header should allow it to be cached;

(3) It does not cause other files to be loaded. For example, an HTML file containing other resources is not suitable.

Finding these resources requires crawling page contents of the target site. To give a lower bound of the feasibility of the attack, we crawl every site's front page for such resources. In addition, we try to find two common resources on the site, *favicon.ico* and *robots.txt*. The former is the small icon image for a website, and the latter is the file to tell web crawlers how the site owner wants them to behave.

We use Alexa top 1000 sites for the experiment. There are 68 sites not returning a valid page visiting *http(s)://foo.com*, where *foo* is the name of each site. For the remaining 932 sites, we find suitable resources on 760 sites. It does not mean that the other 172 sites do not have suitable resources, because our crawling is a rather shallow one, which does not crawl into inner pages and subdomains of the target site.

Attack. The attack has only two differences from the "unsafe port" attack in Section 3.2: (1) instead of loading the URL with an unsafe port, it loads the URL of the resource; (2) before any measurement, the attack script visits the resource once to put the resource in the disk cache. As before, we remove 7 data points (or 1%) as outliers. Figure 12 shows the values of the remaining 753 sites. The BG-NP differentiation is still 100%. The number of FG-BG overlapping is 157 (or 20.85%), and the number of FG-NP overlapping is 82 (or 10.89%).

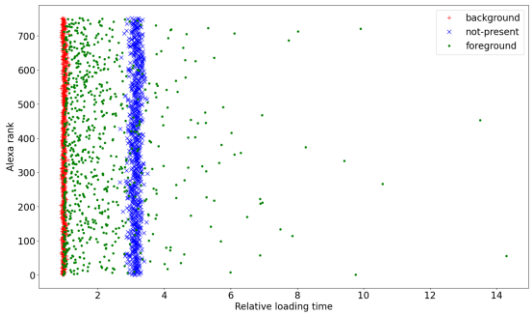
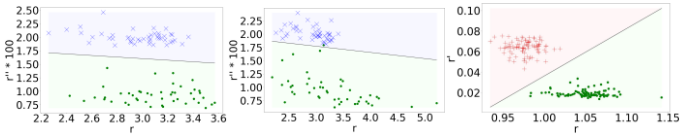


Figure 12: the attack using disk cache

The second run is less convenient than using the unsafe port approach, because the resources are site-specific. When using the unsafe port approach, we conveniently put all the sites together to get an overall distribution. However, when using the site-specific resources, we lose the convenience, and need to get a distribution result for every site. Figure 14 shows three cases we study, each showing 92 data points of a site sampled during a 16-hour duration. The cases show that the measurements r′ and r′′ remain clearly effective when we use the disk-cache mechanism. Admittedly, profiling each site is a substantial inconvenience for the attacker. However, it is a one-time effort, which is not a fundamental mitigating factor of the threat.



gamespot.com (FG-NP) | vnexpress.net (FG-NP) | house365.com (FG-BG)

Figure 14: individual cases of the second run using cached resources

5.3 Evaluation using different browsers

The attack is evaluated using different browsers for two purposes: (1) to present a convincing evidence that the enabler of the attack is indeed the site isolation mechanism; (2) to evaluate the attack on Microsoft Edge and Firefox.

Finding the attack-enabling version of Chromium. There are a great number of changes checked into the Chromium codebase. To experimentally confirm that our attack is enabled by the site isolation mechanism, not by any other changes, we compile and test a wide range of Chromium versions using a "binary search", based on whether a version is vulnerable or not. Figure 13 shows the versions that our attack is tested against. The binary search ends up with a tight range between version 68.0.3402.0 and 68.0.3403.0. The Chromium

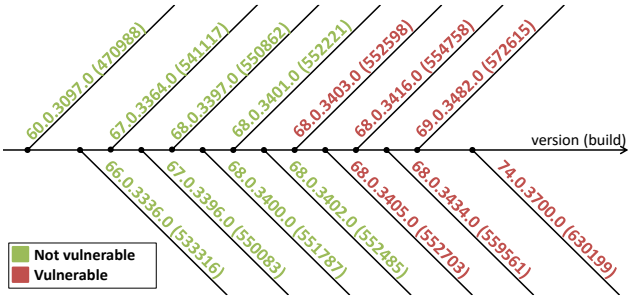


Figure 13: Chromium versions that we test

changelog confirms that site isolation became enabled by default at version 68.0.3403.0 [26].

Microsoft Edge. As Microsoft Edge has been rebuilt based on Chromium in 2019, it also inherits site isolation from Chromium. We test the attack on Edge 83.0.478.56 running on Windows 10, and observe a similar pattern of r, compared with Chrome 83.0.4103.116 running on the same machine and OS. Table 2 shows the values of some top sites. For Microsoft Edge, the r values in the "not present" and "background" situations are also distributed in two narrow stripes, and the r values of the "foreground" situation also reflect their page complexity. These distribution characteristics can be seen in Table 2.

Browser	Site	r_{np}	r_{fg}	r_{bg}
Edge	google.com	4.351909	1.080008	0.973337
	amazon.com	4.276265	2.985395	0.995525
	youtube.com	4.374174	13.482331	1.007836
Chrome	google.com	3.454074	1.352866	1.326693
	amazon.com	3.582909	2.889311	1.140386
	youtube.com	3.887231	13.405612	1.122806

Table 2: Microsoft Edge compared with Chrome

Firefox. Site isolation is currently an experimental feature in Firefox's nightly build. To enable it, the fission.autostart option needs to be set true. Firefox uses a different rendering engine, namely Gecko. As we showed in Sections 3 and 4, our attack is a result of a deep and broad investigation about Chromium's internal. Obviously, the same investigation needs to be conducted for Gecko to see whether our attack can be fully reproduced against Firefox. Although we have not investigated deeply about Gecko, we have already reproduced the first run, which allows the attacker to detect the presence of target sites.

We use the disk-cache approach in this evaluation. We identify 730 sites in Alexa top 1000 sites as the dataset. Figure

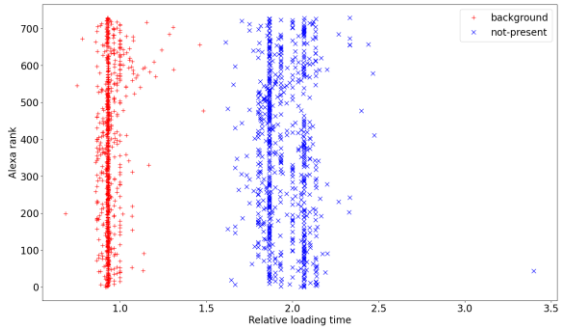


Figure 15: NP and BG values when Firefox's Fission is on

15 shows that all the BG and NP points are separated, without removing any outlier. Compared to Chromium's distribution, there are two differences: (1) the Firefox distributions are wider. We believe that this is due to the experimental nature of the feature. The feature is still premature at this point, as we see serious performance degradation when it is turned on. (2) The precision of Firefox's timing API is reduced to one millisecond [29], as a mitigation against Meltdown and Spectre. This causes the values to be more "quantized" than those in Chromium.

5.4 Batch testing for the Alexa top 500 websites

We evaluate the accuracy of the batching method described in Section 3.4. The target-site set is the Alexa top 500 sites. In every evaluation, we randomly open 5 sites from the set, and run the presence test through the 500 sites to see the accuracy. We repeat the experiment for 104 runs. On average, it takes 56.536 seconds to complete the batch on a test machine with an Intel i7-8565U @ 1.8GHz and 16GB of RAM. Table 3 shows the numbers of runs resulting in 0 to 3 false positives and false negatives. Among the 104 runs, 99 of them (95.19%) produce no false positives, and 86 of them (82.69%) produce no false negatives. We consider these a high accuracy for the one-minute time budget.

Number of errors	0	1	2	3
False positive	99/104	4/104	1/104	0/104
False negative	86/104	9/104	7/104	2/104

Table 3: Results of the batch testing for Alexa top 500 sites

6 Potential Mitigations and Responsible Disclosure

This section discusses potential mitigations, and report our interactions with the Chrome and Edge teams.

6.1 Potential mitigations

The mitigation approaches can be considered under three categories: (1) adding noise or decreasing the precision of time-measurement APIs; (2) finer-grained process boundaries; (3) not reusing processes across tabs. We discuss these approaches below, and emphasize that they are all tradeoffs between effectiveness, performance and functionality.

Mitigation 1: adding noise or decreasing the precision of time-measurement APIs. A reasonable mitigation is to add a random delay during the renderer process creation. A moderate level of random delay will make the FG-NP and the FG-BG detections harder. However, to cause an effective BG-NP overlapping, the delay needs to be hundreds of milliseconds, which may be a difficult tradeoff in practice. Another approach, which does not incur any performance overhead, is to make the time-measurement API less accurate. Earlier in Figure 15, we see that Firefox lowers the API precision to one millisecond to mitigate the Meltdown and Spectre attack. Similarly, Microsoft Edge and Internet Explorer lowered the precision to 20 µs [9]. In our case, the resolution would need to be lowered to tens/hundreds of milliseconds. Again, it is a difficult tradeoff.

Mitigation 2: finer-grained process boundaries. One of the next steps of the site isolation effort is to explore the feasibility of "origin isolation", which offers finer grained process boundaries (see Section 6.3 of Reis et al. [19]). Under "origin isolation", the attacker will need to hypothesize more potential targets, because the number of origins is larger than the number of sites, so the target-sites list needs to be large. "Origin isolation" can be challenging to design and implement, considering its resource and performance requirements.

Mitigation 3: not reusing processes across tabs. A good mitigation would be to prohibit tabs from sharing processes. The challenge, as we explain in Section 2, is the scarcity of processes. We see two usability and programmability issues of this approach. First, it will greatly limit the number of tabs the user can open for sites like CNN.com and Foxnews.com, which contains many cross-site iframe. Second, web technologies are sophisticated. It is necessary for platform companies, e.g., advertisement, social networking, cloud, and CDN companies, to provide web middleware. If the reusing is prohibited, popular middleware will be restrained from using iframes, as it will increase the number of processes in every tab that contains it.

6.2 Responsible disclosure

We reported our findings and shared a version of this paper to Chrome and Edge team in early July 2020. We received confirmation of the issue from the Edge team, and are having several rounds of discussions with the Chrome team and the Edge team on possible mitigations.

The Chrome team initially asked us to evaluate a proposal: to isolate error pages in a separate process. This would make the "unsafe port" technique no longer effective. The proposal motivated us to construct the disk-cache attack described in Section 5.2. The Chrome team acknowledged that the proposal would not completely address the core problem.

We suggest to the Chrome team a mitigation for the FB-BG detection: to add an early exit to blink::SetSelectionColors (which is called by CreateView, as shown in Section 4.4.2). If the selection colors of the newly created RenderView has no change from the renderer's existing settings, it can skip the style recalculation, thus avoid exposing the currently visible frame. Chrome team expressed the intention to consider it.

The Chrome team proposed an idea related to mitigation 3. It was to avoid reusing a process for any top-level frame, but still allow same-site iframes to reuse a process. This would prevent the iframes on the attacker page from residing in the same process as any top-level frame in another tab. We considered this a helpful mitigation. However, the Chrome team admitted that the attacker could fingerprint a victim page based on the sites in all its iframes. Since the proposed mitigation does not prevent the detection of these sites' presence in the iframes, the attacker may still infer the site in the top-level frame.

Status update. The Chrome team gave us an update in July 2021. They had made some progress in putting error pages in different processes. An experiment about the revised process reuse policy, as described in the paragraph above, had been scheduled, but had a dependency on another experiment that the team was undertaking. Therefore, it would take more time to get a conclusive result about the new policy.

7 Related work

Section 2 has described the related literature about site isolation, so we do not include them in this section. There are many research papers about browser's timing channel and other side-channels, summarized below.

Timing-based cross-tab page identification. Using the timing channel to infer a page on another tab has been shown before, which is directly related to our work. Vila and Köpf [22] showed that the event loop of the browser process (called the *host process* in the paper) allowed a timing-based page-identification attack, so that the attacker on one tab could identify the page on another tab. Our work is different from Vila and Köpf's work in three aspects. First, site isolation is expected as a countermeasure of Vila and Köpf's attack, but is the enabler of our attack. Second, the accuracy of our attack is much higher. The accuracy of the cross-tab attack in [22] is 23%, measured using 500 main pages from Alexa's Top sites. Third, our test is conducted with extra tabs rendering unrelated webpages, more realistic than having only the attacker and victim tabs.

Another attempt for cross-tab page identification is by Kim et al. [12]. It tries to use the temporal changes of memory footprint sizes as the side channel to identify the *very next site visited after the attack starts*. Although the attack shows effectiveness under certain assumptions, the authors admit that the assumptions are too friendly to attackers: no background memory usage variation, no extra tab and a small sample set of 100 sites. They agree with other researchers' assessment that these assumptions may be unrealistic.

History sniffing – timing based. History sniffing is the attack to determine if a page corresponding to a URL has been visited by the user before. Cascading Style Sheet (CSS) provides opportunities for this. Smith et al. show that the timing of CSS Paint API can infer the value of the visited property of a URL [21], which indicates whether the URL has been visited before. (The original *CSS-visited* based attacks do not need to exploit the timing channel. Janc and Olejnik have a paper about the basic mechanism [10]. The problem has been fixed by all major browsers.) Kotcher et al. use CSS filters to do image transformations on every pixel in a target region on the screen. The timing of every transformation allows the attacker to determine the approximate color of the pixel [14]. The attacker can thus "see" the color of a link to decide if the URL has been visited. Along the same line of pixel-stealing, Andrysco et al. discover that the multiply operation takes much longer to complete when the operands are *subnormal floating-point numbers*. The SVG filter, a type of CSS filter, can exploit the timing difference to determine if a pixel is white or black [2].

History sniffing – using other side channels. Besides the timing channel, other side channels also provide opportunities for history sniffing. Lee et al. show that GPU's memory is a side channel, because one application's data processed by the GPU can be accessed by another application. As an example, they show that when Chromium and Firefox turn on their GPU-enabling options, an attacker program can infer whether a URL has been visited before, based on the "webpage texture" left in the GPU memory [15]. As mentioned above, paper [10] references several early *CSS-visited* based attacks. They are not based on timing channels.

Other browser side-channel studies. There are other side channel consequences and studies in the literature. Lee et al. discover that the new HTML5 functionality, *AppCache*, allows a cross-site attacker to determine the login status of the victim user, and an attacker outside a local network to determine if a URL exists in the local network [16]. The AppCache side channel is independently discovered by Goethem et al. [6], who demonstrate the inference of certain user account attributes in various social networks. Bortz et al. show that cross-site timing can potentially reveal a user's login status or the size of a user's hidden data (e.g., the approximate number of items in the shopping cart) [4]. Sanchez-Rola et al. showed a timing-based attack to enumerate the extensions installed in a browser, thus fingerprint the browser for the user tracking purpose [20]. Besides the local side channels, researchers also study how to reduce the network jitter when measuring the timing characteristics from a remote machine. A recent work shows that the relative timing of concurrent requests is more robust against the network jitter than the traditional absolute timing [7]. Note that, our attack, either using the unsafe port or the disk cache mechanism, avoids the network jitter.

8 Conclusions

We show that site isolation enables a robust timing attack, which allows the attacker's script to test which target sites are loaded into the browser, and determine whether each loaded site is in a foreground tab that interacts with the user. The attack is a result of an in-depth investigation and experimentation about a wide range of timing characteristics involving cross-site interactions under site isolation. We get very high vulnerability percentages about this issue, when evaluating it using the Alexa Top 3000 websites. We also show that the attack can do the presence test for the 500 sites altogether within roughly one minute, achieving a high accuracy. The attack does not need any special assumption, thus can be turned into a realistic privacy threat in the general population.

The main challenge revealed by our work is the tension between the scarcity of processes and the objective of site isolation to separate cross-site frames in different processes. We are working with the Google Chrome team and Microsoft Edge team to propose and evaluate mitigation options.

Acknowledgments

We thank the anonymous reviewers for valuable feedbacks for improving the paper. Michael Ens, Johnathan Norman, Charlie Reis and Andrew Ritz of the Chrome and Edge teams offered many detailed technical insights and discussions. Xian Zhang and Xuan Feng helped improve the presentation. We also thank Mia Chen for producing a video demo included in this paper.

References

[1] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. Towards a Formal Foundation of Web Security. Proceedings of the 23rd IEEE Computer Security Foundations Symposium, 2010

[2] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, Hovav Shacham. On Subnormal Floating Point and Abnormal Timing. Proceedings of the IEEE Symposium on Security and Privacy (Oakland), May 2015

[3] Adam Barth, Collin Jackson, Charles Reis, and Google Chrome Team. The Security Architecture of the Chromium Browser. http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf, 2008.

[4] Andrew Bortz, Dan Boneh, Palash Nandy. Exposing Private Information by Timing Web Applications. Proceedings of the International conference on World Wide Web (WWW), 2007

[5] Martin Brinkmann. Firefox 70: Site Isolation testing begins officially. https://www.ghacks.net/2019/08/11/firefox-70-site-isolation-testing-begins-officially/

[6] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In the Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015.

[7] Tom Van Goethem, Christina Pöpper, Wouter Joosen, Mathy Vanhoef. Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections. Proceedings of the USENIX Security Symposium, 2020.

[8] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the OP web browser. Proceedings of the 2008 IEEE Symposium on Security and Privacy (Oakland), May 2008.

[9] John Hazen. Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer. https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/

[10] Artur Janc, Lukasz Olejnik. Web Browser History Detection as a Real-World Privacy Threat. In Proceedings of European Symposium on Research in Computer Security 2010

[11] Yaoqi Jia, Zheng Leong Chua, Hong Hu, Shuo Chen, Prateek Saxena, Zhenkai Liang. The "Web/Local" Boundary Is Fuzzy: A Security Study of Chrome's Process-based Sandboxing. Proceedings of the ACM Conference on Computer and Communications Security (CCS), October 2016

[12] Hyungsub Kim, Sangho Lee, Jong Kim. Inferring browser activity and status through remote monitoring of storage usage. Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC), 2016

[13] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In the Proceedings of the 40th IEEE Symposium on Security and Privacy, 2019.

[14] Robert Kotcher, Yutong Pei, Pranjal Jumde, Collin Jackson. Cross-Origin Pixel Stealing: Timing Attacks Using CSS Filters. Proceedings of the ACM Conference on Computer and Communications Security, 2013

[15] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. Proceedings of the IEEE Symposium on Security and Privacy (S&P), 2014.

[16] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Identifying Cross-origin Resource Status Using Application Cache. Proceedings of the Network and Distributed System Security Symposium (NDSS), 2015.

[17] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading Kernel Memory from User Space. In the Proceedings of USENIX Security, 2018.

[18] Charles Reis, Steven Gribble. Isolating Web Programs in Modern Browser Architectures. Proceedings of the 4th ACM European conference on Computer systems (EuroSys), April 2009

[19] Charles Reis, Alexander Moshchuk, Nasko Oskov. Site Isolation: Process Separation for Web Sites within the Browser. In the Proceedings of the 28th USENIX Security Symposium. August, 2019. Santa Clara, CA, USA

[20] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. "Extension breakdown: Security analysis of browsers extension resources control policies." In the Proceedings of the 26th USENIX Security Symposium, 2017

[21] Michael Smith, Craig Disselkoen, Shravan Narayan, Fraser Brown, Deian Stefan. Browser history re:visited. In Proceedings of USENIX Workshop on Offensive Technologies (WOOT), 2018

[22] Pepe Vila, and Boris Köpf. "Loophole: Timing attacks on shared event loops in chrome." Proceedings of the 26th USENIX Security Symposium (USENIX Security 17). 2017.

[23] Helen Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, Herman Venter. The multi-principal OS construction of the gazelle web browser. In Proceedings of the 18th conference on USENIX security symposium, August 2009

[24] The Chromium Projects: Process Models. https://www.chromium.org/developers/design-documents/process-models

[25] Cross-document message – HTML Standard. https://html.spec.whatwg.org/multipage/web-messaging.html#web-messaging

[26] Log - 68.0.3402.0..68.0.3403.0 - chromium/src - Git at Google. https://chromium.googlesource.com/chromium/src/+log/68.0.3402.0..68.0.3403.0?pretty=fuller&n=10000

[27] Microsoft Edge: Making the web better through more open source collaboration Windows Experience Blog. https://blogs.windows.com/windowsexperience/2018/12/06/microsoft-edge-making-the-web-better-through-more-open-source-collaboration/

[28] HTML Standard, https://html.spec.whatwg.org/#origin

[29] Firefox API performance.now(). https://developer.mozilla.org/en-US/docs/Web/API/Performance/now

[30] Video demo about the attack. https://youtu.be/vdu81XYX2Ew

Appendix: datasets (b), (c), (d)

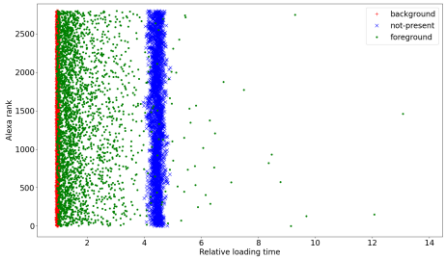


Figure 16: loading time distribution, dataset-(b)

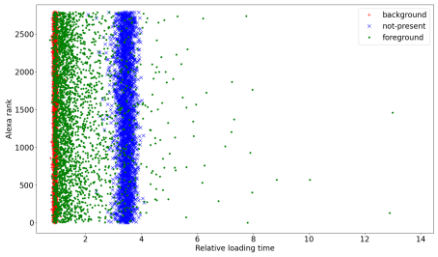


Figure 20: loading time distribution, dataset-(c)

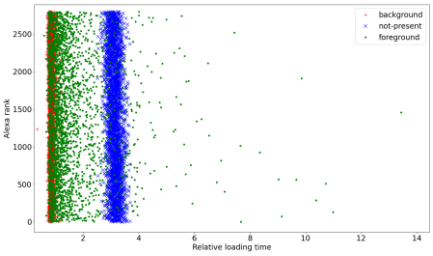


Figure 24: loading time distribution, dataset-(d)

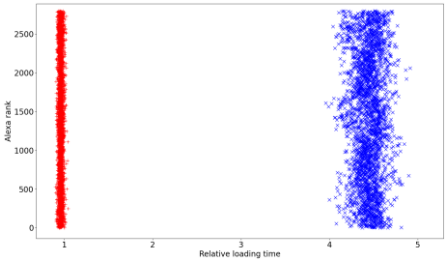


Figure 17: Background vs not-present, dataset-(b)

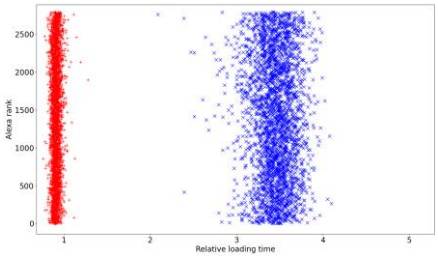


Figure 21: Background vs not-present, dataset-(c)

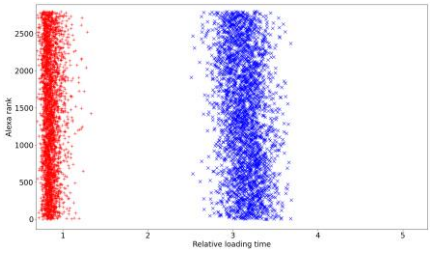


Figure 25: Background vs not-present, dataset-(d)

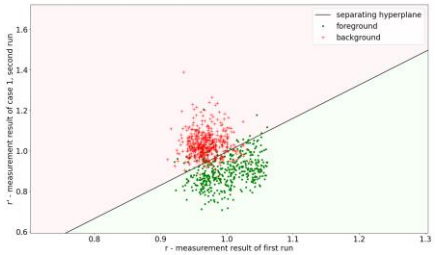


Figure 18: foreground vs background, dataset-(b)

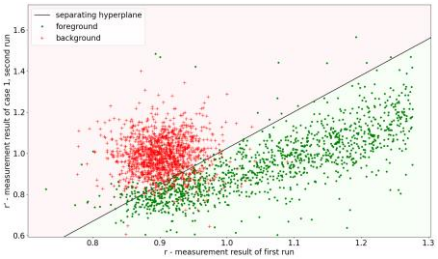


Figure 22: foreground vs background, dataset-(c)

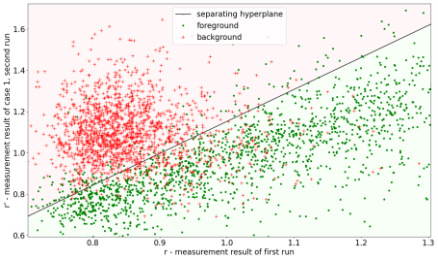


Figure 26: foreground vs background, dataset-(d)

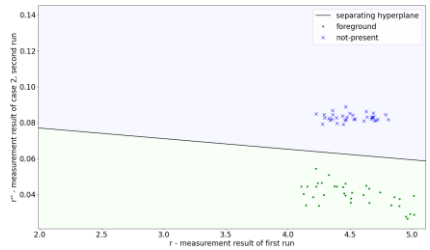


Figure 19: foreground vs not-present, dataset-(b)

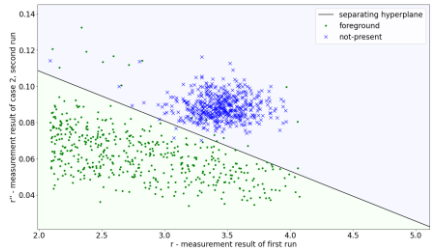


Figure 23: foreground vs not-present, dataset-(c)

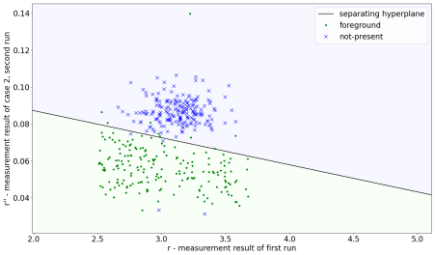


Figure 27: foreground vs not-present, dataset-(d)