THE UNIVERSITY
of ADELAIDE

Browser-Based Microarchitectural Side-Channel Attacks

Sioli O'Connell

A thesis submitted for the degree of
DOCTOR OF PHILOSOPHY
The University of Adelaide

Supervised by
Yuval Yarom and
Damith Ranasinghe

January 2025

Contents

List of Figures

List of Listings

List of Tables

Abstract

Web browsers have become a critical component of the modern computing ecosystem. They execute code from websites to enable rich interactions; however, this capability can be exploited by malicious websites to launch attacks directly on user devices. The risk is further amplified by microarchitectural side-channel attacks, which leverage hardware characteristics to leak sensitive data. Although comprehensive theoretical countermeasures exist, they are often impractical for use across entire browsers. As a result, browser vendors have resorted to implementing ad-hoc countermeasures to address these threats.

This issue raises the central question of this thesis: *Are these ad-hoc countermeasures effective in protecting users against microarchitectural side-channel attacks?* To answer this question, the thesis investigates and implements microarchitectural variants of four attack types: website fingerprinting, pixel stealing, memory disclosure, and reduced-round encryption attacks.

The thesis begins by investigating the underlying causes of leakage in three recent microarchitectural website-fingerprinting attacks. The findings reveal that multiple independent sources contribute to the observed leakage, each leaking sufficient information to enable website fingerprinting. These results suggest that effective protection requires comprehensive and multi-faceted countermeasures.

The thesis then introduces two attacks: Pixel Thief and Spook.js, both of which are practical, end-to-end microarchitectural attacks implemented in JavaScript and capable of targeting modern browsers. Pixel Thief is a cache-based pixel-stealing attack that leverages data-dependent memory access patterns in Scalable Vector Graphics filters to recover portions of rendered webpages. Spook.js is a memory disclosure attack that exploits transient type confusion to access arbitrary process memory. Together, these attacks demonstrate that previous mitigation efforts against microarchitectural threats are insufficient.

Finally, the thesis presents a proof-of-concept (PoC) attack against *controlled leakage* in security type systems through a reduced-round encryption attack on the Advanced Encryption Standard (AES). Security type systems enable developers to annotate secret values, allowing the compiler to automatically enforce protections against leakage. These systems often assume sequential execution, however modern processors exhibit out-of-order execution. The PoC attack exploits this mismatch in execution semantics to leak secret values by triggering controlled leakage earlier than the developer intended.

While this thesis shows that ad-hoc countermeasures have been insufficient, it does not claim they are ineffective. The attacks presented here have had reduced impact, required more sophisticated implementation techniques, and required stronger assumptions of adversarial capabilities demonstrating the efficacy of these countermeasures. Furthermore, this work has also informed browser vendors and website operators in the development of new countermeasures that further reduce the threat posed by microarchitectural attacks.

Statement of Originality

I certify that this work contains no material which has been accepted for the award of any other degree or diploma in my name, in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that no part of this work will, in the future, be used in a submission in my name, for any other degree or diploma in any university or other tertiary institution without the prior approval of the University of Adelaide and where applicable, any partner institution responsible for the joint award of this degree. The author acknowledges that copyright of published works contained within the thesis resides with the copyright holder(s) of those works.

I give permission for the digital version of my thesis to be made available on the web, via the University's digital research repository, the Library Search and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

I acknowledge the support I have received for my research through the provision of an Australian Government Research Training Program Scholarship.

Acknowledgements

I dedicate this thesis to my family and friends, both academic and personal. I genuinely appreciate each and every one of you. Thank you for joining me on this long, difficult, but rewarding journey. Without you, none of this would have been possible.

– Cheers, Sioli

I would like to further thank the following people:

My supervisors Yuval and Damith (and unofficial supervisor Daniel)

My co-authors

My friends at UofA, UMich and Georgia Tech

Finally, my parents Shane and Margaret

Chapter 1

Introduction

Over the past thirty years, the Internet has become an integral part of modern life, revolutionising communication by connecting billions of people to each other and to the services they use daily.

The primary gateway to the Internet is the browser, software that connects to and renders websites. Given their central role in connecting users to online services, browsers have become a critical component of the modern computing ecosystem. People rely on them to access everything from routine content to highly sensitive information. As a result, browsers accumulate vast amounts of sensitive data, making them attractive targets for adversaries.

Malicious websites attack browsers by exploiting how browsers, by default, automatically execute code embedded within webpages. This code, typically written in JavaScript, is designed to enable rich, interactive user experiences. However, the same capabilities can be misused by adversaries to launch attacks from within the browser.

Fortunately, browser vendors are aware of these threats, and modern browsers are designed to isolate websites from one another and from the rest of the system. A key component responsible for maintaining this isolation is the JavaScript engine, which executes JavaScript code within the browser. The JavaScript engine enforces several security-critical invariants, such as bounds and type checking, and enables the browser to mediate access to external resources, including the file system, network, and other websites.

The rest of the browser security model is built upon this foundation. If the JavaScript engine fails to maintain its security guarantees, a malicious website may be able to launch attacks against the broader system. This thesis investigates attacks where the JavaScript

engine fails to maintain its security guarantees because the hardware itself failed to maintain its security guarantees.

These attacks exploit optimisations in modern processors that enhance performance by adapting to patterns observed in past program behaviour. These optimisations create a flow of information: from program behaviour to the processor's internal state, and subsequently to the execution time of future operations. Microarchitectural attacks reverse this flow by measuring a program execution time, which reveals constraints on the processor's internal state and, in turn, on the possible past behaviours of a victim program. With enough constraints, an adversary can infer sensitive information processed by the victim.

This thesis focuses on two types of microarchitectural attacks: cache timing attacks and transient execution attacks. The first cache timing attack on browsers was demonstrated by Oren et al. (2015), who adapted an earlier technique for recovering memory access patterns in victim programs (Osvik et al., 2006; Liu et al., 2015) to the browser context. They used this capability to detect network and mouse activity from other websites and other programs running on the system.

The first transient execution attack was published by Kocher et al. (2019). They demonstrated several techniques for manipulating branch prediction in modern processors and showed how this ability could be used to recover sensitive information in various scenarios. One such scenario involved bypassing a bounds check in the JavaScript engine to trigger an out-of-bounds memory access.

These discoveries posed serious challenges for browser vendors, as the security of browser isolation mechanisms assumed the hardware could maintain its security guarantees. Both cache timing and transient execution attacks revealed that these guarantees are not maintained in practice. In, response, browser vendors implemented countermeasures to mitigate the threat of these attacks, including cross-origin isolation (MDN Contributors, 2024b,a) and site isolation (Reis et al., 2019).

Cross-origin isolation limits a website's ability to simultaneously access high-resolution timers and embed third-party content, effectively forcing malicious websites to choose between embedding sensitive third-party content *or* executing cache attacks to recover that content. Site isolation places each website in a separate process to mitigate the threat of transient execution attacks. While transient execution attacks can still access arbitrary process memory, the accesses remain confined to the isolated process.

However, state-of-the-art attack techniques have progressed since the deployment of these countermeasures, prompting the central question of this thesis:

> *Are cross-origin isolation and site isolation effective techniques to prevent state-of-the-art microarchitectural attacks on browsers?*

This thesis addresses that question by presenting several novel and practical attacks that bypass these countermeasures to recover sensitive information from modern browsers. The remainder of this chapter provides further introduction to each chapter of the thesis, summarises its key contributions, and outlines its structure.

1.1 Analysing Coarse-Grained Side-Channel Leakage

A key challenge in mounting microarchitectural side channels is that they depend on the precise behaviour of the microarchitecture, a behaviour that is rarely documented. Therefore, researchers must rely on experimentation to gather data and construct theoretical models that align with the observed behaviour. The goal is that these models are accurate enough to allow us to design effective countermeasures by accurately predicting the behaviour of the microarchitecture under a variety of conditions.

This approach has proven remarkably effective when applied to individual microarchitectural components, with successful attacks targeting memory caches Liu et al. (2015); Osvik et al. (2006); Percival (2005); Yan et al. (2019); Yarom and Falkner (2014), branch predictors Evtyushkin et al. (2016); Acıiçmez et al. (2007, 2006); Evtyushkin et al. (2018); Zhang et al. (2020), translation lookaside buffers Gras et al. (2018); Koschel et al. (2020); van Schaik et al. (2018), shared buses Paccagnella et al. (2021); Wan et al. (2022), execution units Acıiçmez and Seifert (2007); Bhattacharyya et al. (2019); Aldaya et al. (2019), and GPUs Wei et al. (2020); Naghibijouybari et al. (2018); Taneja et al. (2023); Cronin et al. (2021); Owens and Wang (2011), However, this methodology becomes less effective when analysing the behaviour of the system as a whole.

Beyond the inherent complexity of modelling an entire system, accurately attributing observed leakage to specific microarchitectural effects remains a major challenge. This challenge is evident in three recent microarchitectural website-fingerprinting attacks: cache occupancy (Shusterman et al., 2019), loop counting (Cook et al., 2022), and

mwait (Zhang et al., 2023). These attacks identify websites based on site-specific patterns in overall system performance. While their success demonstrates that the system leaks information, pinpointing the exact source of the leakage has been a significant challenge.

In particular, disagreement exists over the source of leakage observed in the cache occupancy attack. Shusterman et al. (2019) attribute the leakage to cache contention, while Cook et al. (2022) suggest interrupts cause the leakage. *Which explanation, if any, is correct?*

Gülmezoglu (2021) proposes a method for attributing leakage to specific browser behaviours. This approach involves instrumenting the browser to log its behaviour and correlating these logs with microarchitectural activity using hardware performance counters. A machine learning model is then trained to perform website fingerprinting, and explainable AI techniques are used to determine which browser behaviours most influence the model predictions. While this method effectively links leakage to browser-level activity, it does not clarify how information flows through specific microarchitectural channels – a critical detail for designing robust and effective countermeasures.

Cook et al. (2022) take a different approach by controlling various microarchitectural channels to manage leakage within the system. They conduct experiments across multiple system configurations to determine which microarchitectural channels contribute to the observed leakage. While this method helps identify contributing channels, limited control leaves uncertainty about whether the detected leakage stems from controlled channels or from uncontrolled leakage elsewhere.

Contributions. Chapter 3 introduces a methodology that addresses these gaps and resolves the conflicting explanations for leakage in the cache occupancy attack.

The chapter begins by identifying four microarchitectural channels commonly discussed in the literature: contention for on-core resources, cache contention, interrupts, and frequency scaling. It then demonstrates methods to control each of these channels, resulting in the complete elimination of leakage for the loop counting and mwait primitives, and a significant reduction of leakage for the cache occupancy primitive. These results indicate that the four channels comprehensively explain leakage in the loop counting and mwait cases, while additional uncontrolled channels contribute to leakage in the cache occupancy case. The chapter further investigates these uncontrolled channels through experiments suggesting that the remaining leakage originates off-core, likely

within components of the memory hierarchy (Paccagnella et al., 2021; Dai et al., 2022; Wan et al., 2022; Pessl et al., 2016; van der Veen and Gras, 2023).

With comprehensive control established for the loop counting primitive, the chapter proceeds to identify which of the four channels contribute to leakage. It demonstrates that each channel independently provides enough information to mount a website fingerprinting attack, indicating that models and countermeasures focusing on a single channel are incomplete.

To present a more complete picture of leakage, the chapter concludes by quantifying the relative contribution of each channel to the total observed leakage. This quantification is achieved by selecting hardware performance counters that correlate strongly with leakage through specific channels and calculating their correlation with observations from each primitive. The results reveal that the primary contributors to leakage are cache contention for the cache occupancy primitive, frequency scaling for loop counting, and interrupts for mwait, thereby resolving the conflicting explanations for leakage in the cache occupancy attack.

1.2 Mounting a High-Capacity Pixel-Stealing Attack

Webpages often include content from multiple sources – the main webpage itself, links highlighted based on the user browsing history, third-party advertisements, social media widgets, and more. Browsers are responsible for rendering all this content as a seamless, cohesive page without leaking sensitive information between the different sources.

One of the key mechanisms browsers use to achieve this is the same-origin policy, which restricts code from one website from accessing content belonging to another website, even if both are embedded within the same webpage. Beyond controlling direct access, browsers also control indirect access including controlling whether code can access the composited output that is displayed to the screen.

Unfortunately, pixel-stealing attacks bypass this control to reveal sensitive information displayed on webpages. Several studies have demonstrated how to reveal the colour of selected pixels by measuring subtle differences in webpage rendering times (Stone, 2013; Kotcher et al., 2013; Andrysco et al., 2015; Kohlbrenner and Shacham, 2017; Wang et al., 2023; Taneja et al., 2023). By repeating this process with different pixels, an adversary can reconstruct arbitrary portions of the webpage.

A fundamental limitation of these attacks is their dependence on measuring the time taken to render the webpage, which restricts the information extraction rate to the browser rendering frequency – typically 60 times per second. Furthermore, browser vendors have actively worked to eliminate these timing differences to mitigate previously published attacks. Although recent studies have shown that CPU and GPU frequency scaling can be exploited to conduct pixel-stealing attacks, these methods result in even lower leakage rates (Wang et al., 2023; Taneja et al., 2023).

Beyond the limitations on the rate of information extraction, browser vendors have widely deployed cross-origin isolation to mitigate a broad range of side-channel attacks. Cross-origin isolation creates a mutually exclusive condition where a website must choose between embedding content from other sources *or* accessing high-resolution timers. As a result, a malicious website cannot simultaneously embed sensitive content *and* use high-resolution timers to mount an attack to recover that content.

Contributions. Chapter 4 addresses the question: *Are high-capacity pixel-stealing attacks on modern browsers feasible?* It presents Pixel Thief, a cache-based pixel-stealing attack that bypasses the rate limit imposed by the browser rendering frequency.

Pixel Thief achieves this by observing data-dependent memory accesses within the `feComponentTransfer` filter. This filter is applied to sensitive content, an action allowed because the side effects of applying filters are assumed to be unobservable. The filter uses the colour of each pixel in the sensitive content to access a table of colours. The output of the filter is ignored, but its memory accesses are observed using the Prime+Probe technique. Each observed memory access reveals the colour of the pixel used to perform the access. Since the filter performs an access for each pixel, the colour of several pixels can be leaked during a single invocation of the filter. This decouples the leakage rate of Pixel Thief from the browser rendering frequency and overcomes the limitations that constrained previous attacks.

To measure filter memory accesses with Prime+Probe, the chapter introduces a generic technique to bypass cross-origin isolation. This technique splits the attack across two webpages: the first embeds sensitive content, while the second uses high-resolution timers to measure leakage from the first webpage.

The chapter concludes by presenting two end-to-end attacks to demonstrate the effectiveness of Pixel Thief's approach. The first adapts a method from Stone (2013) to modern fonts to increase leakage rates by exploiting redundant information in text,

then uses this method to reveal the victim's Wikipedia identity. The second abuses link highlighting to construct an oracle that reveals whether the user previously visited a specific URL, then queries the oracle with a large number of URLs to partially reconstruct the victim's browsing history.

1.3 Mounting a Transient-Execution Attack on Modern Browsers

The discovery of transient execution attacks sent shockwaves through the industry. Browsers were particularly vulnerable, as they used software-based isolation techniques to separate content rendered onto the same webpage. Unfortunately, transient execution attacks bypass many software-based techniques by enabling adversaries to transiently execute code before critical security requirements are verified.

In response, Google deployed site isolation for the Chrome browser (Reis et al., 2019). This countermeasure separates content from different origins into separate processes. While it does not prevent transient execution attacks from accessing arbitrary memory within a process, it mitigates the threat of transient execution attacks by leveraging process isolation guarantees provided by the hardware.

Beyond hardware-based isolation, Chrome employs pointer compression, representing each pointer as a 32-bit offset from a base address. This reduction in pointer size saves memory but also restricts the memory accessible to each pointer. Chrome leverages this limitation, combined with a specialised allocator, to partition the address space and confine out-of-bounds memory accesses.

Contributions. Chapter 5 addresses the question: *Are practical transient-execution attacks on modern browsers feasible?* The chapter presents Spook.js, a transient-execution attack capable of extracting sensitive information despite the countermeasures deployed by Chrome.

The chapter begins by highlighting a gap between policies used by the browser to define the security boundaries between websites. Typically, the same-origin policy defines the boundary between two websites and is often the boundary developers consider when designing websites. Site isolation, however, uses the same-site policy to define the boundary between two websites. This gap creates a scenario where websites considered

distinct by developers are combined into the same process undermining site isolations security benefits.

To bypass Chrome's address space partitioning and access memory from other websites, the chapter demonstrates a type-confusion attack which tricks Chrome into accessing a malicious object as a typed array. Since typed arrays can be shared between webpages, Chrome allocates them outside of any specific partition and uses a full-sized pointer to refer to the contents of the typed array. Spook.js crafts a malicious object that takes control over this pointer to access memory anywhere in the process.

To demonstrate the practicality of Spook.js, the chapter presents several attack scenarios grouped by adversarial capabilities. First, scenarios where the adversary controls a webpage (e.g., a user homepage) and recovers sensitive content displayed on the same site. Next, scenarios where the adversary uploads content to a cloud service and retrieves other user data stored on that service. Finally, scenarios where the adversary installs a malicious extension on the victim's browser and extract data stored in other extensions.

The chapter concludes by revealing that other browsers, particularly Chrome-based browsers such as Chromium, Brave, and Edge, are also vulnerable to Spook.js with minimal adaptation.

1.4 Transient-Execution Attacks on Security Type Systems

Cryptographic constant-time programming is a widely used technique to mitigate side-channel attacks. It involves transforming a program so that its observable behaviour remains consistent regardless of any secret data. Typically, this transformation is performed manually, which can be both time-consuming and error-prone

One way to automate this process is through the use of a security type system, a programming language feature that enables developers to annotate variables in their programs with **secret** or **public** labels. Tools such as the FaCT compiler (Cauligi et al., 2019) then use type checking and information-flow analysis to ensure data stored within secret variables can never leak into public variables.

While this approach is effective, it can be overly restrictive. It prohibits any side effect from depending on secret variables, even when such dependencies are intentional and

do not pose a security risk. Consider, for example, an encrypted messaging application. Encryption takes a message and a key as inputs, both of which are stored in secret variables to protect their integrity. The output is an encrypted message and is safe to reveal. It is stored in a public variable so that it can be sent over the network. However, because the encrypted message is derived from the secret key and message, the compiler will reject the program.

A common solution to this problem is to introduce a **declassify** operation. This operation serves as an annotation inserted by the developer to assert that a value obtained from a secret variable is safe to reveal. In the example above, the developer would insert a declassify operation after encrypting the message so that it can be sent over the network.

Contributions. Chapter 6 addresses the question: *Can values in secret variables unintentionally leak even when declassification is applied correctly?*.

The chapter answers the question in the negative, demonstrating a PoC attack on several implementations of AES, including industry-standard versions and those protected by the FaCT compiler. The attack exploits branch prediction and out-of-order execution in modern processors to coerce the processor into declassifying an output before encryption is correctly applied. Since the output is declassified, the compiler allows it to be revealed.

The chapter provides details for the PoC attack while the accompanying paper (Shivakumar et al., 2023) provides the full theoretical analysis, methods to recover the key from recovered ciphertexts, and proposed countermeasures to mitigate the issue.

1.5 Summary of Contributions

In summary, the contributions of this thesis are as follows:

Analysing Coarse-Grained Website-Fingerprinting

- Describes how to control microarchitectural channels to eliminate leakage (Section 3.2).
- Establishes that all identified channels contribute to website fingerprinting attacks (Section 3.3).

- Quantifies leakage through each channel across multiple website finger printing attacks (Section 3.4).

Mounting a High-Capacity Pixel-Stealing Attack

- Develops a simple method to bypass cross-origin isolation, enabling the use of high-resolution timers alongside embedded third-party content (Section 4.1).
- Introduces Pixel Thief, a cache-based pixel-stealing attack that exploits content-dependent memory access patterns in the `feComponentTransfer` SVG filter to achieve data extraction rates exceeding the browser rendering frequency (Sections 4.2 and 4.3).
- Adapts techniques for rapid text recovery using pixel stealing (Section 4.4).
- Demonstrates a fast history-sniffing attack leveraging cache-based pixel stealing (Section 4.5).

Mounting a Transient-Execution Attack on Modern Browsers

- Introduces Spook.js, a transient memory-disclosure attack capable of reading arbitrary memory within rendering processes (Section 5.1).
- Examines the limitations of site isolation and identifies conditions under which multiple websites are consolidated into the same process (Section 5.2).
- Investigates the security implications of Spook.js on Chrome extensions (Section 5.3).
- Confirms that other Chromium-based browsers, including Microsoft Edge and Brave, are also vulnerable to Spook.js (Section 5.4).

Transient-Execution Attacks on Security Type Systems

- Demonstrates PoC attacks on protected implementations of AES (Chapter 6).

Open Source Code Releases
As part of this thesis, the following open-source code releases have been made:

- *https://github.com/0xADE1A1DE/PixelThief*
- *https://github.com/spookjs/spookjs-poc*

1.6 Structure of Thesis

Chapter 2 – Background – provides the essential background on microarchitectural attacks. It offers a general overview of the behaviour of key microarchitectural components, explains how measurement primitives exploit this behaviour to extract information, reviews the history of attacks based on these primitives, and discusses various browser features relevant to each attack.

Chapter 3 – Analysing Coarse-Grained Side-Channel Leakage – presents and evaluates a methodology for investigating leakage in coarse-grained side channels.

This chapter has not yet been published. In this work, I developed the approach to control channels and eliminate leakage, designed the method to establish leakage through each channel, and supervised experiments conducted by a co-author.

Chapter 4 – Mounting a High-Capacity PIxel Stealing Attack on Modern Browsers – presents and evaluates a practical, high-capacity attack that recovers portions of rendered webpages.

This chapter is based on the following publication: Sioli O'Connell, Lishay Aben Sour, Ron Magen, Daniel Genkin, Yossi Oren, Hovav Shacham, and Yuval Yarom – "Pixel Thief: Exploiting SVG Filter Leakage in Firefox and Chrome", USENIX Security 2024.

In this work, I developed the architecture to bypass cross-origin isolation, identified data-dependent behaviours in SVG filters, and designed and implemented Pixel Thief to exploit these behaviours. I also designed the text-stealing attack, supervised its implementation by a co-author, and designed and implemented the history-sniffing attack.

Chapter 5 – Mounting a Transient Execution Attack on Modern Browsers – demonstrates the first practical transient execution attack on browsers and evaluates it under several attack scenarios.

This chapter is based on the following publication: Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom – "Spook.js: Attacking Chrome Strict Site Isolation via Speculative Execution", IEEE SP 2022.

In this work, I designed and implemented the Spook.js attack, including using type confusion to break out of Chrome's partitioned address spaces, identifying suitable malicious objects, preventing deoptimisation through nested speculation, widening the speculation window via last-level cache evictions, and porting Spook.js to other Chromium-based browsers.

Chapter 6 – Declassification and Transient Execution – explores the interaction between security type systems and transient execution.

This chapter has been published as Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom – "Spectre Declassified: Reading from the Right Place at the Wrong Time", IEEE SP 2023.

In this work, I developed the PoC attack used to extract partial ciphertexts from OpenSSL and FaCT implementations of AES.

Chapter 7 – Conclusions – summarises the results, discusses their implications for browser security, and outlines directions for future research.

1.6.1 Other Publications

The main body of this thesis includes publications to which I made significant contributions. Other publications in which my role was more limited are have been published as follows:

Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom – "Prime+Probe 1, Java-Script 0: Overcoming Browser-based Side-Channel Defenses", USENIX Security 2021.

Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom – "BunnyHop: Exploiting the Instruction Prefetcher", USENIX Security 2023.

Bradley Morgan, Gal Horowitz, Sioli O'Connell, Stephan van Schaik, Chitchanok Chuengsatiansup, Daniel Genkin, Olaf Maennel, Paul Montague, Eyal Ronen, and Yuval Yarom –

"Slice+Slice Baby: Generating Last-Level Cache Eviction Sets in the Blink of an Eye", IEEE SP 2025.

Chapter 2

Background

This chapter provides essential background for the rest of the thesis. It is divided into four sections, the first two detail the internal behaviour of processors and techniques that exploit these behaviours to measure internal processor state, while the last two focus on relevant features of web browsers and browser-based attacks.

2.1 Microarchitecture

The *microarchitecture* refers to the internal design of a processor that determines how it executes instructions. It includes components such as caches, buffers, pipelines, execution units, interconnects, and other internal structures. While processors may share the same Instruction Set Architecture (the interface between the hardware and software), their microarchitectures can differ significantly.

Unfortunately, processor vendors rarely publish detailed documentation of microarchitectural behaviour, which has led various research communities to reverse-engineer these details to better understand how microarchitectures operate. The rest of this section provides a concise overview of modern processor microarchitectural behaviours relevant to this thesis.

2.1.1 Memory Caches

Modern processors use a cache – small, fast memory located directly on the processor – to reduce the average latency of memory accesses. This improvement is achieved partly

through physical proximity and partly because caches are optimised for low latency at the expense of data density and power efficiency. The processor stores recently accessed data in the cache under the assumption that this data will be reused. When the data is accessed again, the lower latency of the cache results in faster access times, leading to overall performance improvements.

Structure. The cache is organised as a *set-associative* structure, consisting of multiple *sets*, each containing multiple *ways*. The processor divides memory into fixed-length units called *cache lines*, which are stored in the cache. Each cache line can only be stored in a single set, determined uniquely by the address of the line, but it can occupy any way within that set. To identify which line is stored in each way, the cache stores the line address (or tag) alongside the data. When the processor performs a memory access, it uses the memory address to determine the set to search, then searches each way for a matching address. If a match is found, the access is a *cache hit*, and the data from that way is returned. If no match is found, the access is a *cache miss*, and the processor instead retrieves the data from system memory or a higher-level cache.

Hierarchy. Modern caches are not only set-associative but also organised into a hierarchy. Lower-level caches (closer to the processor cores) typically have lower latency at the expense of a smaller capacity, while higher-level caches have larger capacity at the expense of a higher latency. Usually, the *last-level* cache is shared among all processor cores, whereas the lower-level caches are private to each core.

Inclusivity. Some caches are *inclusive*, meaning that lower-level caches contain a strict subset of the data stored in higher-level caches. That is, any data present in an inclusive lower-level cache must also be present in higher-level caches. The processor maintains this inclusive property by ensuring that when data is evicted from a higher-level cache, all-inclusive lower-level caches also evict that data.

Other caches exhibit an *exclusive* property, where lower-level caches do not contain any data that is also stored in higher-level caches. In this configuration, when data is moved from a higher-level cache to an exclusive lower-level cache, the processor preserves exclusivity by evicting the data from the higher-level cache. Consequently, each piece of data exists in only one level of the cache hierarchy at a time.

Caches that are neither inclusive nor exclusive are referred to as *non-inclusive*. Like

inclusive caches, data copied to a non-inclusive lower-level cache is retained in the higher-level caches. However, unlike inclusive caches, when data is evicted from the higher-level caches, non-inclusive lower-level caches will continue to retain a copy of the data.

Replacement Policy. Since sets have a fixed number of ways, they can become full. When the processor needs to store a new line in a full set, it makes room in the set by evicting another line and storing that line back in system memory. The line selected for eviction is determined by the cache replacement policy. A commonly used policy is the pseudo-least-recently-used (pLRU) policy, which approximates the behaviour of a true least-recently-used (LRU) policy while requiring fewer resources than a faithful implementation.

2.1.2 Execution

To improve performance, modern processors employ both *out-of-order* execution and *superscalar* execution. Out-of-order execution allows the processor to execute instructions as soon as their dependencies are resolved, rather than strictly following the original program order. Superscalar execution enables each core within the processor to execute multiple instructions in parallel, rather than processing one instruction at a time. Together, these features enhance performance by allowing the processor to hide execution delays and keep its execution units busy with independent instructions.

Branch Prediction. Branches limit the effectiveness of superscalar out-of-order execution because the outcome of a branch determines which instructions should be executed next. As a result, all future instructions become implicitly dependent on the branch condition, preventing their execution until the condition is known. Processors remove this implicit dependency by predicting the branch outcome and executing the program under the assumption the prediction is correct. When program execution matches the prediction, the processor saves time since it did not have to wait for the branch condition. In cases where program execution does not match the prediction, the processor is said to have *mispredicted*. To correct the mispredicted branch, the processor '*squashes*' (discards) program execution following the branch, then restarts execution with the correct branch outcome.

2.2 Microarchitectural Attacks

Microarchitectural side-channel attacks exploit the behaviour of the microarchitecture to extract sensitive information from other processes running on the same machine. These attacks leverage unintended links between program secrets and the program's internal behaviour. By observing that behaviour – typically through measurements of contention on shared microarchitectural resources – an attacker can infer those secrets.

In its simplest form, an attack might involve timing how fast a loop executes (Cook et al., 2022). However, often attacks exploit the nuanced behaviour of individual microarchitectural components to induce contention at finer granularities, thereby revealing more detailed information. Research has demonstrated that the behaviour of many microarchitectural components can be exploited to mount side-channel attacks. These components include memory caches Liu et al. (2015); Osvik et al. (2006); Percival (2005); Yan et al. (2019); Yarom and Falkner (2014), branch predictors Evtyushkin et al. (2016); Acıiçmez et al. (2007, 2006); Evtyushkin et al. (2018); Zhang et al. (2020), translation lookaside buffers Gras et al. (2018); Koschel et al. (2020); van Schaik et al. (2018), shared buses Paccagnella et al. (2021); Wan et al. (2022), execution units Acıiçmez and Seifert (2007); Bhattacharyya et al. (2019); Aldaya et al. (2019), or GPUs Wei et al. (2020); Naghibijouybari et al. (2018); Taneja et al. (2023); Cronin et al. (2021); Owens and Wang (2011). The remainder of this section provides background on the techniques used to mount the attacks described in this thesis, along with a brief overview of transient-execution attacks.

2.2.1 Cache Timing Attacks

The fundamental idea behind a cache timing attack is that an adversary can infer whether a specific piece of memory is present in the cache by measuring the time it takes to access that memory. Fast access times (typically under a hundred CPU cycles) suggest that the data was served from the cache, meaning the memory was present in the cache at the time of access. Conversely, slower access times suggest that the memory was not cached and had to be fetched from a higher level of the memory hierarchy.

On the surface, revealing whether memory is cached does not appear to expose sensitive information. However, cache state is influenced by memory accesses made by all programs running on the machine and the patterns of those memory accesses are often

correlated with sensitive information. By measuring cache state, a cache timing attack can partially reveal these memory access patterns and, in turn, reveal sensitive information.

This thesis mainly employs the Flush+Reload and Prime+Probe techniques to measure memory access patterns. In both techniques, one party accesses memory (the victim), while the other attempts to detect these accesses (the adversary). Although these roles are traditionally referred to as 'victim' and 'adversary' in the literature, this thesis adopts alternative terminology: accessor and observer. This naming scheme is chosen to emphasise the broader applicability of these techniques beyond conventional cross-process attack scenarios. For example, the techniques can be used in reverse engineering, in situations where the accessor and observer are components of the same program, or as building blocks within larger attacks where both roles are performed by the adversary.

Flush+Reload. In the Flush+Reload technique (Gullasch et al., 2011; Yarom and Falkner, 2014) the observer starts by flushing the target memory from the cache. The accessor is then allowed to execute, potentially accessing the target memory. After a short interval, typically a few thousand processor cycles, the observer measures the time it takes to access the target memory. Fast access times indicate the memory is stored in the cache, while slower access times suggests absence from the cache.

The primary advantage of Flush+Reload is its ease of use and low noise levels. However, its main limitation is that the accessor and observer must share memory; more precisely, the observer must be able to access the same memory used by the accessor. This requirement significantly restricts Flush+Reload's applicability in adversarial settings simply because the adversary is often unable to access the target memory. Moreover, Flush+Reload depends on specific instructions to evict memory from the cache, further limiting its use to cases where the adversary can execute arbitrary instructions.

Prime+Probe. The Prime+Probe technique (Osvik et al., 2006; Liu et al., 2015) overcomes the limitations of Flush+Reload by exploiting contention within cache sets. The observer starts by filling every way of a target cache set with 'junk' data. After the accessor runs, the observer measures the time it takes to access this junk data again. A longer access time indicates that the junk data was likely evicted from the cache, likely because the processor removed it to make room for data accessed by the accessor, causing the access to be served from slower system memory.

The primary advantages of Prime+Probe are that the observer does not need to share

memory with the accessor and does not require special instructions. However, there are two significant challenges to effectively using Prime+Probe.

The first challenge is identifying which junk data to access, known as the (minimal) eviction set. The eviction set consists of enough memory to fill a target cache set – no more, no less. Throughout this thesis, we rely on the excellent method developed by Vila et al. (2019) to find eviction sets.

The second challenge is locating the target cache set to fill, the specific set that serves memory to the accessor. Typically, the observer employs a brute-force approach: building an eviction set for each cache set and then using an attack-specific detection method to determine whether accesses to a given eviction set collide with the accessor's accesses.

Beyond these challenges, Prime+Probe is slower than Flush+Reload because it needs to access an entire eviction set, whereas Flush+Reload only needs to access a single piece of memory. In addition, Prime+Probe yields less precise information than Flush+Reload, as it detects contention caused by any memory access served from the target cache set, not just the specific memory location of interest.

2.2.2 Transient-Execution Attacks

Transient-execution attacks are microarchitectural attacks that exploit the execution of instructions that may later be squashed. In these attacks, the adversary tricks the processor into executing the program incorrectly, causing behaviour that deviates from the original program. This often includes accessing memory the program normally would not be able to access. Although the processor eventually squashes the architectural state associated with this incorrect execution, there is often a delay between incorrectly executing an instruction and squashing its effects. This delay is called the speculation window and it enables an adversary to execute code while the program is in an invalid state. Because the hardware does not revert all microarchitectural state, the adversary can use the speculation window to encode secrets into the microarchitectural state, then later recover those secrets by measuring residual microarchitectural state.

Inducing Branch Mispredicitons. This thesis mainly focuses on incorrect program execution caused by branch mispredictions. The technique was first described by Kocher et al. (2019), in which the adversary trains a target branch by repeatedly causing it to be taken. When the processor encounters the same branch in the future, it will predict that

the branch will be taken. After this training step, the adversary moves to an attack step where they provide a malicious input. This input would cause the branch to be not taken, but because the processor predicted the branch as taken, it proceeds to take the branch anyway.

Consider, for example, a program that uses a branch to guard an access into an array. Under normal execution, whenever the access is performed the branch verified the provided index was within bounds. However, in a transient-execution attack, this assumption does not hold. The adversary can supply an out-of-bounds index and induce the processor to take the branch. The processor then executes the subsequent instructions, performing an out-of-bounds access reading arbitrary process memory.

Although the processor eventually squashes this incorrect speculative execution, the adversary can still leak the secret through a side channel. Commonly, secrets are leaked through the cache due to well-understood behaviour and numerous established attack techniques.

2.3 Browsers

A web browser is a piece of software that allows users to access and interact with the World Wide Web. Its primary function is to render websites so users can view and engage with their content. Websites consist of various resources identified by a Uniform Resource Locator (URL). To retrieve a resource, the browser connects to a server specified by the URL hostname and port number, then requests the resource using the Hypertext Transfer Protocol (HTTP), referencing the URL path. Some of these resources are webpages, files written in Hypertext Markup Language (HTML) that define the page structure, content, and visual appearance.

2.3.1 Same-Site Policy

When two resources interact, the web browser consults a security policy to determine whether the interaction is permitted. Commonly, the browser enforces the same-origin policy, which allows interaction only when the resources share the same origin – defined by the domain name, protocol (such as HTTP or HTTPS), and port number of the server.

Notably, the entire domain is considered. For example, `foo.example.com` and `bar.example.com` are different origins because their subdomains, `foo` and `bar`, are different.

Websites can opt into different security policies, usually via HTTP Headers – metadata sent along with each resource. One such policy is the same-site policy, which permits interaction between resources served from the same site. Unlike the same-origin policy, the same-site policy considers only the final portion of the domain.

Specifically, the same-site policy considers the Effective Top-Level Domain Plus One (eTLD+1). The Top-Level Domain refers to the final portion of a domain under which websites can be registered (e.g., .com.au or .net). The Effective Top-Level Domain includes domains that allow others to register subdomains (e.g., .github.io or .canva.site). The Effective Top-Level Domain Plus One refers to the domain immediately preceding the effective top-level domain, the part registered by website owners (e.g., example.com or example.github.io).

2.3.2 JavaScript

To enable rich interactive applications, webpages can reference executable resources that browsers download and run on the user's device. These executable resources are typically written in JavaScript – a high-level, dynamically typed, object-oriented programming language featuring prototypal inheritance.

Objects. Objects are mutable, user-defined collections of properties. Each property has a name, which must be a string or a symbol, and a value, which can be of any type, including another object. Properties can be added, modified, or removed after the object is created.

Objects inherit behaviour in JavaScript through *prototypal inheritance*. Each object has a special property called its prototype. When a property is accessed on an object but is not found, the JavaScript engine looks for that property on the object prototype instead. Since the prototype may be an object too, the JavaScript engine repeats this process until either the property is found or until there are no more prototypes to check.

Dynamic Typing and Speculative Optimisations. In JavaScript, variables do not have fixed types. Instead, any value of any type can be assigned to any variable, even if that variable previously held a value of a different type. JavaScript engines reduce the cost of this dynamic behaviour by employing *speculative optimisations* (Meurer, 2017) which optimise the code under the assumption that variables only store one type of value. This

optimisation commonly occurs at the function level, where the engine records argument types each time the function is called. After several calls, the engine uses this recorded type information to generate optimised machine code for the function. However, since future arguments may have different types, the engine inserts checks to verify that the actual argument types match the speculated types. If the types do not match, the engine will discard the optimised code and fall back to a more generic, slower implementation.

Hidden Classes. In addition to the challenges of dynamic typing, objects pose their own challenge since properties can be dynamically added or removed. JavaScript engines typically use hashmaps to support this dynamic behaviour, but each property lookup becomes an expensive hashmap lookup. To reduce this overhead, JavaScript engines use *hidden classes* to create a static layout for object properties then cache property lookups as a speculative optimisation (Deutsch and Schiffman, 1984; Chambers et al., 1989; Hölzle et al., 1991).

Hidden classes store properties in two data structures – an array holding the property values and a hashmap mapping properties to their corresponding locations within the array. When a property is added to a hidden class, the engine appends the property value to the end of the array and updates the hashmap to map the property name to the corresponding array location. When looking up a property, whether to read or modify it, the engine uses the hashmap to find the location of the property value in the array. Although this approach still requires a hashmap lookup, the location of the property is shared between all objects with the same hidden class, which enables the engine to cache the lookup as a speculative optimisation.

2.3.3 Browser Architecture

Modern web use often involves visiting multiple webpages simultaneously, each in its own window or tab. Historically, browsers employed a monolithic single-process architecture that handled all browser functions within one process. This architecture is simple, but it suffers from security and stability issues. For example, if a webpage causes the browser to crash, the entire browser crashes affecting unrelated webpages. Likewise, if a malicious webpage exploits a vulnerability in the browser, then the entire browser becomes compromised.

To address this issue, browser vendors adopted a multi-process architecture consisting of several renderer processes and a single compositor process (Reis et al., 2009; Nguyen, 2017). If a webpage causes the browser to crash, only the affected renderer process crashes while the rest of the browser continues unaffected. Similarly, if a malicious webpage exploits a vulnerability, only the affected renderer process becomes compromised.

Partitions in Chrome. To reduce the memory footprint of renderer processes, the Chrome browser compresses pointers by representing them as 32-bit offsets from a fixed base address (Sheludko and Solanes, 2020). This base address defines an *isolate* within which memory is allocated. Because pointers are represented as 32-bit offsets, instead of a full 64-bit address, the amount of memory accessible to each pointer is limited. Chrome leverages this limitation as a security feature by carefully arranging the address space so that memory accesses through compressed pointers are constrained to their respective isolate.

2.3.4 `Uint8Array`

Typed arrays are a collection of standard objects that enable developers to directly access and manipulate raw binary data using JavaScript. All typed array data is stored within an `ArrayBuffer` or a `SharedArrayBuffer`. These buffer objects do not provide an interface for direct data access; instead, one or more typed array views can be instantiated over the buffer to provide an array-like interface. For example, a `Uint8Array` exposes its underlying buffer as an array of 8-bit unsigned integer values.

Whenever a JavaScript program accesses a typed array, the engine first verifies that the index is valid – specifically, that the index is an integer greater than or equal to zero and less than the array length. If the index is invalid, the engine does not perform any memory access.

Implementation in Chrome. In Chrome, typed arrays contain four key fields: the type, a pointer to the underlying buffer, the length of the underlying buffer, and an offset into the underlying buffer (used to implement zero-copy slice operations). To support inexpensive moving and sharing of typed arrays between webpages and JavaScript threads, the underlying buffer is allocated at an arbitrary address outside of any given isolate. Consequently, references to the underlying array buffer are represented as full-sized

pointers. Figure 2.1 illustrates the memory layout for `Uint8Array`'s implemented in Chrome.

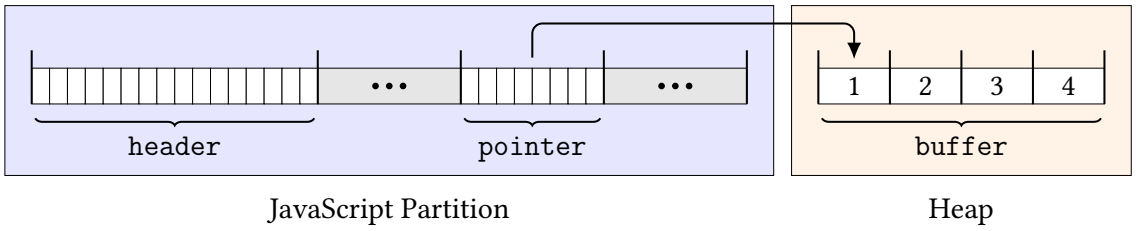


FIGURE 2.1: **Uint8Array Memory Layout:**
A `Uint8Array` containing the values $1, 2, 3, 4$. Ticks denote the boundaries of fields, rectangles within each field denote a byte, and colours are used to highlight where memory is allocated.

2.4 Browser-Based Attacks

Web browsers are commonly targeted owing to the popularity of the web as a platform and the ease with which an adversary can execute code on a victim's device. Once an adversary tricks a victim into visiting a malicious website, through phishing, malicious advertisements, or other means, they gain the ability to execute code on the victim's device. From there, the adversary can mount a variety of attacks against other open webpages, the browser itself, or the underlying system. Of particular interest to this thesis are microarchitectural attacks launched from the browser (Oren et al., 2015; Shusterman et al., 2019, 2021; Andrysco et al., 2015, 2018). This section provides a brief introduction to website-fingerprinting, history-sniffing, and pixel-stealing attacks.

2.4.1 Website-Fingerprinting Attacks

In a website-fingerprinting attack, the adversary attempts to uncover the identity of a website visited by the victim. There are two broad attack models for website-fingerprinting attacks: the on-path and co-located models.

In the more conventional on-path model, the adversary mounts their attack from a separate machine located somewhere along the communication path between the victim and the wider internet (Hintz, 2002; Herrmann et al., 2009; Panchenko et al., 2011; Cai et al., 2012; Gong et al., 2012; Wang and Goldberg, 2013; Juarez et al., 2014; Hayes and Danezis, 2016; Panchenko et al., 2016; Rimmer et al., 2018; Jansen et al., 2018; Li et al., 2018). This position enables the adversary to monitor network packets sent to and from the victim. Since the contents of each packet are assumed to be encrypted, the adversary is limited to only using packet size and the timing between packets to fingerprint a website. Consequently, countermeasures against on-path attacks typically focus on injecting random delays or spurious traffic to mask these patterns.

Under the co-located model, the adversary mounts their attack directly from the victim device. This position enables the adversary to indirectly monitor the browser behaviour through side channels including microarchitectural side channels (Oren et al., 2015; Shusterman et al., 2021, 2019; Cronin et al., 2021; Naghibijouybari et al., 2018; Taneja et al., 2023; Cronin et al., 2021).

In either model, the adversary samples browser behaviour over time to construct a fingerprint then matches it with previously recorded fingerprints to identify the website.

2.4.2 History Sniffing Attacks

In a history-sniffing attack, the adversary aims to reveal the victim's browsing history. The defining characteristic of this attack is that the adversary cannot directly access the history itself. Instead, they rely on an oracle that reveals whether a given URL exists in the victim's browsing history then they query the oracle with a large number of URLs to partially reconstruct the victim's browsing history.

Many history-sniffing attacks exploit the `:visited` CSS selector, a usability feature in browsers that allows pages to change the style of visited links. Using this method, the adversary creates a page containing multiple links with URLs they wish to query. The adversary then checks which links have been styled to reveal which URLs have been visited by the victim. Early attacks could directly read which styles were applied to a link or could use styles that would alter the page layout to indirectly reveal whether the style had been applied (Smith et al., 2018; Janc and Olejnik, 2010). In response, browser vendors deployed countermeasures that remove differences between visited and unvisited

link styles. Specifically, whenever a website attempts to query the browser, the browser will always behave as though the URL is unvisited. In cases where the style makes this impossible, the browser will simply refuse to apply the style at all.

More recent history-sniffing attacks have bypassed these countermeasures by exploiting timing side channels. Specifically, the adversary creates a style that is computationally expensive for the browser to apply and uses the `:visited` selector to apply this style. By monitoring for performance degradation, the adversary can infer whether the style was applied (Smith et al., 2018). In response, browser vendors further restricted allowed styles to a limited set believed to have low performance overhead, such as changing the text colour or background colour of links (MDN Contributors, b).

2.4.3 Pixel-Stealing Attacks

Pixel-stealing attacks are timing attacks that induce data-dependent delays in browser rendering to reveal sensitive information displayed on the screen. These attacks apply a malicious style, designed to be expensive to compute for certain colours but cheap for others, to an HTML element to create a delay dependent on the element colour. If the difference in computation time is large enough, the browser misses the deadline to render the next frame, an event that an attack can detect to reveal the original element colour.

SVG Filters. Attacks typically use SVG filters to create computationally expensive styles. These filters are small user-defined functions (originally specified by the SVG standard) that are intended to enable artistic effects, such as blurring, which are difficult or impossible to achieve with standard HTML and CSS alone. Filters are composed of several filter elements, primitive operations that the webpage provides parameters for, that are combined together to form a complete filter. The list of filter elements is specified by the SVG standard and includes several standard image filtering operations such as colour mapping and convolutions.

Listing 2.1 demonstrates how to apply a filter to a `div` element. Lines 3-8 define an SVG object. Inside the object, Lines 4-7 define the filter. This filter consists of two elements executed sequentially: A Gaussian blur (Line 5) and image dilation (Line 6). Finally, Line 11 applies the filter to the target `div` element.

Stealing Cross-Origin Content. Filters can be applied to cross-origin content to allow designers to create webpages with cohesive filtered appearances. This is assumed to

```
<html>
<head>
  <svg><defs>
    <filter id="filter_id">
      <feGaussianBlur stdDeviation="1" />
      <feMorphology operator="dilate" radius="2" />
    </filter>
  </defs></svg>
</head>
<body>
  <div style="filter: url(#filter_id)"><!-- content --></div>
</body>
</html>
```

LISTING 2.1: **SVG Filter Example:**
A demonstration of how to define an SVG filter and apply it to a specific element on a webpage.

be safe because the browser restricts access to the output of the filter. Unfortunately, the presence of side-channels invalidates this assumption and enables adversaries to recover cross-origin content. Previous studies has demonstrated how differences caused by algorithmic optimisations in SVG filters (Stone, 2013; Kotcher et al., 2013), computations on subnormal floating-point numbers (Andrysco et al., 2015; Kohlbrenner and Shacham, 2017), and frequency scaling in modern CPUs and GPUs (Wang et al., 2023; Taneja et al., 2023) can be exploited to mount pixel-stealing attacks.

Chapter 3

Attributing Microarchitectural Leakage within Systems

Several approaches exist for analysing microarchitectural side-channel leakage, with the most common method being to reverse-engineer a model of the behaviour of a specific hardware component and then develop an attack that exploits that behaviour. The successful execution of the attack is taken as evidence that the reverse engineering is accurate, and the resulting model is used to provide insight into the leakage. Many studies have employed this approach for various components, including memory caches (Liu et al., 2015; Osvik et al., 2006; Percival, 2005; Yan et al., 2019; Yarom and Falkner, 2014), branch predictors (Evtyushkin et al., 2016; Acıiçmez et al., 2007, 2006; Evtyushkin et al., 2018; Zhang et al., 2020), translation lookaside buffers (Gras et al., 2018; Koschel et al., 2020; van Schaik et al., 2018), shared buses (Paccagnella et al., 2021; Wan et al., 2022), execution units (Acıiçmez and Seifert, 2007; Bhattacharyya et al., 2019; Aldaya et al., 2019), and GPUs (Wei et al., 2020; Naghibijouybari et al., 2018; Taneja et al., 2023; Cronin et al., 2021; Owens and Wang, 2011).

This approach works well when analysing information leakage from specific components; however, it is less suited to analysing coarse-grained leakage. Such leakage occurs in attacks that monitor the system over extended periods of time or monitor large portions of the microarchitectural state. In either case, it can be difficult to attribute leakage to any singular microarchitectural effect. The challenge is further compounded by the use of machine learning classifiers to extract information, as it further obscures which factors contributed to a successful attack.

As an example of this challenge, consider the cache-occupancy website-fingerprinting

attack. In this attack, the adversary measures the speed at which they can repeatedly iterate though a large buffer of memory to infer which website a victim visits. Successful cache-occupancy attacks demonstrate that information must be leaked, but Shusterman et al. (2019) and Cook et al. (2022) provide competing explanations for the mechanism behind this leakage.

Shusterman et al. (2019) propose that browser memory activity causes evictions in the adversary's buffer which, in turn, affects the adversary's iteration frequency. Cook et al. (2022) question this explanation and argue that operating system interrupts better correlate with the adversary's iteration frequency.

Several studies have begun to tackle this problem. Gülmezoglu (2021) proposed an approach that uses explainable AI techniques to reveal which browser behaviours leak the most information. Specifically, they annotate browser behaviour and record it alongside HPCs. Then, they train a machine learning model to distinguish websites based on the HPCs and use explainable AI to identify which measurements – and by extension, which browser behaviours – provided the most information to the model. While this approach effectively highlights which browser behaviours leak information to an adversary, it falls short of explaining how the leaked information propagates through the microarchitecture to the attacker, insight that is crucial for designing effective countermeasures.

Cook et al. (2022) themselves proposed an approach that reveals how information leaks through the system by performing experiments while controlling microarchitectural channels. Specifically, they isolate programs using operating system interfaces and inject noise into specific microarchitectural components. Then, they conduct experiments across multiple system configurations and measure the effect that these configurations have on an attack. If the attack performs similarly well, then it is assumed that the controlled channels must not leak any information. However, they do not fully control all channels, leaving open the possibility that information leaks through uncontrolled channels.

This chapter addresses these gaps by presenting a method that comprehensively controls information leakage through microarchitectural channels. Section 3.1 begins with a description of the primitives underlying three recent website-fingerprinting attacks: the cache-occupancy attack in question, the loop-counting variant proposed by Cook et al. (2022), and a third attack proposed by Zhang et al. (2023). Section 3.2 enumerates four commonly discussed microarchitectural channels and describes methods to completely control information flow (for some attacks). Section 3.3 leverages this control to investigate

each channel and finds that information flows through all four channels. Finally, Section 3.4 examines the relative contribution of each channel to provide insight into the primary causes of leakage for each primitive – resolving the conflicting explanations for the cache-occupancy attack.

While the approach presented in this chapter is used in the context of website-fingerprinting attacks, I hope that it can serve as a basis for a more generic approach to analysing coarse-grained leakage in the future.

3.1 Analysis Overview

This section provides an overview of the measurement primitives, explains how measurements from these primitives are collected and reported, and describes the hardware and software used in the experiments presented in this chapter.

3.1.1 Measurement Primitives

Listing 3.1 shows pseudocode for the measurement primitives used in the cache occupancy, loop counting, and mwait attacks. Each primitive counts the number of operations it can perform within a fixed period. The cache-occupancy primitive counts the number of times it can advance a pointer through a linked list, the loop-counting primitive simply counts the number of loop iterations, and the mwait primitive count the number of times it can call the `wait` operation.

The cache-occupancy primitive employs a linked list that covers the size of the last-level cache. Each element in the list occupies a full cache line and contains pointers to the next and previous elements (Listing 3.2). The order of the elements in the list is randomised to minimise the effects of prefetching. As an optimisation, the first and last elements are linked together to enable the cache-occupancy primitive to iterate through the list multiple times without needing special handling for the end of the list.

This work differs from previous approaches, which iterate through the entire list in every loop iteration (Cook et al., 2022; Shusterman et al., 2019, 2021). Instead, we advance the pointer one element in the list for every loop iteration, ensuring comparable levels of precision across all primitives.

```
T = 5 milliseconds
pointer = create_list()

for each sample {
  count = 0

  start = time()
  while (time() - start < T) {
    pointer = *pointer
    count = count + 1
  }

  samples[sample] = count
}
```

(A) Cache Occupancy

```
T = 5 milliseconds


for each sample {
  count = 0

  start = time()
  while (time() - start < T) {

    count = count + 1
  }

  samples[sample] = count
}
```

(B) Loop Counting

```
T = 5 milliseconds


for each sample {
  count = 0

  start = time()
  while (time() - start < T) {
    wait()
    count = count + 1
  }

  samples[sample] = count
}
```

(C) mwait

LISTING 3.1: **Psuedocode for Measurement Primitives**
Psuedocode for the three measurement primitives studied in this chapter. Each primitive shares a similar structure which measures the number of times a specific action can be performed – a memory access (**Cache Occupancy**), no action (**Loop Counting**), and waiting using a specialised instruction (**MWAIT**). Highlighted lines show the lines that differ between each primitive.

```
// Total size: 64 bytes -- 1 cache line
struct Element {
    struct Element* next;       // 8 bytes
    struct Element* previous;   // 8 bytes
    uint8_t padding[42];        // 42 bytes
};
```

LISTING 3.2: **Linked-List Element**

3.1.2 Measurement Collection

In the following section, we explain our approach for obtaining traces and presenting the resulting data.

Recording a Trace. Before we record a trace, we visit *example.com* to ensure we have a consistent starting point across all traces. Once the page has fully loaded, we start executing the target primitive and record its measurements to a buffer. We begin recording measurements before navigating to the target website to guarantee that the trace captures the beginning of each page load. After 50 milliseconds, we navigate to the target website.

We record 3,000 samples, capturing each sample 5 milliseconds apart, resulting in a total collection time of 15 seconds. This duration was chosen to ensure the page was fully loaded, even when we lower the speed of the processor (Section 3.2.3).

Collecting Traces. Before collecting any traces, we visit each target website, allowing it to fully load and remain displayed for 30 seconds. This ensures that all traces are gathered with a consistent browser cache state and better represent real-world scenarios, where a user is likely revisiting a site.

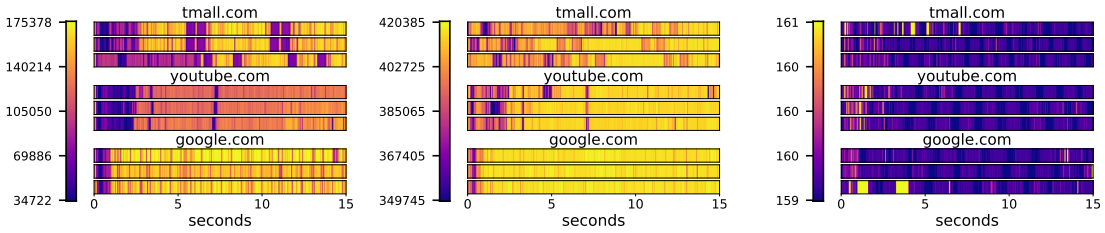
We collect 100 traces for each of the Alexa Top 10 websites. To avoid bias from the collection order, we randomise the sequence in which websites are visited. The trace collection process is automated with Python, which controls the browser using Selenium while running the measurement primitive in parallel.

Training a Classifier. To identify a website from a recorded trace, we train a random forest classifier using 80% of the data for training and 20% for testing. For each reported

value, we repeat the training process 20 times and report the mean and standard deviation of the classifier accuracy.

In this work, we use this classifier as an indicator of system leakage and do not focus on the precise accuracy value. If the classifier can identify the correct website significantly better than random chance, we conclude that the system leaks information. Similarly, if the classifier cannot identify the correct website, we conclude that the system likely does not leak information.

Measuring Time. We measure time using the `rdtscp` instruction, which counts *reference cycles*. References cycles occur at a fixed rate regardless of the processor's current frequency (Intel, Vol. 3B §18.17). We convert 5 milliseconds into a fixed number of reference cycles in advance and wait for that number of cycles to elapse.



(A) Cache Occupancy
Machine: i7-7700
Accuracy: 96.3±0.8%

(B) Loop Counting
Machine: i7-7700
Accuracy: 93.4±0.8%

(C) MWAIT
Machine: i9-12900K
Accuracy: 47.7±2.2%

FIGURE 3.1: **Baseline:**
Measurements of the cache occupancy, loop counting, and mwait primitives on a baseline system visualised as heatmaps.

Heatmaps. Throughout this work, we visualise the collected data using one-dimensional heatmaps, following the style of Shusterman et al. (2019). We apply a linear colour map with the minimum and maximum values set to the 2.5% and 97.5% percentiles of the data, respectively. While this approach removes outliers and makes visual features in each heatmap easier to identify, it also means that colour scales differ between figures.

Figure 3.1 illustrates baseline heatmaps for each primitive. The horizontal axis represents samples over time, and the colour at each position corresponds to the number of

iterations of the inner loop. Lighter colours indicate more iterations, while darker colours indicate fewer iterations. The vertical axis has no meaning beyond making it easier to view the figure.

Hardware Performance Counters. Hardware performance counters (HPCs) track processor performance metrics such as instruction cycles, cache hits, cache misses, branch mispredictions, and more (Intel, Vol. 3B §20). These counters can be accessed directly from user space using the `rdpmc` instruction. In experiments that use HPCs, we record their values after each 5-millisecond measurement interval.

3.1.3 Experiment Setup

Table 3.1 summarises the hardware and software configurations of the two machines used in the experiments.

Table 3.1: **System Configurations.**

CPU	Intel i9-12900KF	Intel i7-7700
Memory	4× 16 GB DDR5 4800 MT/s	1× 4 GB DDR3 1333 MT/s
Motherboard	MSI Z690-A WIFI	ASUS B150M-A D3
Storage	480 GB Western Digital SATA SSD	120 GB Patriot Burst SATA SSD
OS	Ubuntu 22.04.1 LTS	Ubuntu 22.04.1 LTS
Kernel	Linux 6.2.0	Linux 6.1.38
Browser	Chrome 113.0.5672.63	Chromium 109.0.5414.74
Python	Python 3.10.12	Python 3.10.12
Selenium	Selenium 3.141.0	Selenium 3.141.0
C Compiler	GCC 11.4.0	GCC 11.5.0

3.2 Controlling Channel Contributions

Before analysing how information flows through microarchitectural channels, we first ask: *Which microarchitectural channels exist, and how can we control them?* We focus on four

well-known channels: competition with co-resident threads for execution time (Lampson, 1973), interrupt handling (Cook et al., 2022), frequency scaling (Wang et al., 2022, 2023; Taneja et al., 2023), and eviction from the cache hierarchy (Liu et al., 2015).

In the rest of the section, we describe how we configure the system to control information flow through each channel and examine the effects of this configuration on the cache-occupancy, loop-counting, and mwait primitives. Afterwards, we conclude that we can completely control leakage measured both the loop-counting and mwait primitives but can only partially control leakage measured by the cache-occupancy primitive.

3.2.1 Intracore Contention

To control intracore contention, we isolate the primitive from all other programs running on the machine. Specifically, we designate one core as the 'measurement' core and restrict the primitive execution to this core while forcing all other processes to run on the remaining cores. We achieve this isolation using the Linux kernel parameter `isolcpus`, which prevents the kernel from automatically scheduling any threads on the specified core. We then use the `taskset` utility to explicitly bind the primitive to that core during execution. Although disabling hyperthreading is not strictly necessary, owing to `isolcpus`, we disable it to simplify configuration and verification by ensuring that each core runs only a single thread.

3.2.2 Interrupt Handling

Isolating cores with `isolcpus` also shifts most interrupt handling to non-isolated cores. Throughout this work, we may need to manually reassign which cores handle interrupts. We accomplish this reassignment using the `irqbalance` utility and by configuring the `/proc/irq/[id]/smp_affinity` interface.

This approach controls the effects of movable interrupts; however, some interrupts remain immovable. To control the effects of immovable interrupts, we prevent them from occurring altogether.

Tickless. To remove timer interrupts, we enable *tickless* mode for the kernel. In this mode, timer interrupts are disabled on cores that have only a single thread scheduled for execution. To activate tickless mode, we compiled the kernel with the `CONFIG_NO_HZ_`

`FULL` compile-time option and add the measurement core to the list of cores allowed to enter tickless mode using the `nohz_full` kernel parameter.

Tickless Constraints. To prevent timer interrupts from being scheduled, we impose the following three constraints on each measurement primitive. First, the primitives must never execute a system call that triggers a timer interrupt; specifically, they must avoid any system calls that cause the thread to sleep. Second, the primitives must be single-threaded. Finally, no other processes should be scheduled on the same core as the primitive. Violating any of these constraints may cause the system to schedule timer interrupts.

Tickless Verification. We verify the first constraint through manual code inspection, ensuring that the main loop of each primitive never executes any operation that could cause control to return to the kernel. In cases where `sleep` is required for a specific duration, we replace the sleep call with a busy loop that uses the `rdtscp` instruction to track elapsed time. The second and third constraints are verified using the `/sys/kernel/debug/sched/debug` interface, to ensure that only a single thread is running on the measurement core.

RCU Callback Offloading. Read-Copy-Update (RCU) is a synchronisation mechanism that allows multiple kernel threads to concurrently read from a shared data structure. Callbacks can be registered to execute when a thread releases its reference to that structure. To minimise the latency of the operation releasing the last reference, another core will be interrupted to handle the callback. To remove these interrupts, we add the measurement core to the list of cores that are excluded from RCU callback offloading with the `rcu_nocbs` kernel parameter.

Verifying Absence of Interrupts. We verify the removal of interrupts by monitoring the `/proc/interrupts` and `/proc/softirqs` interfaces. These interfaces report the total number of interrupts handled by the kernel since power-on, categorised by interrupt type and the core that processed them. In addition, we use the `CPU_CLK_UNHALTED.RING0_TRANS` performance counter to track the number of transitions from user to kernel mode. Together, these tools allow us to determine whether any interrupts occurred during trace recording.

We find that in the vast majority of experiments (99.52%), no interrupts occur. In the rare instances where interrupts do occur, they are few and appear uncorrelated with

victim activity. Whenever we report results from experiments without interrupts, we exclude any traces where interrupts were detected.

3.2.3 Frequency Scaling

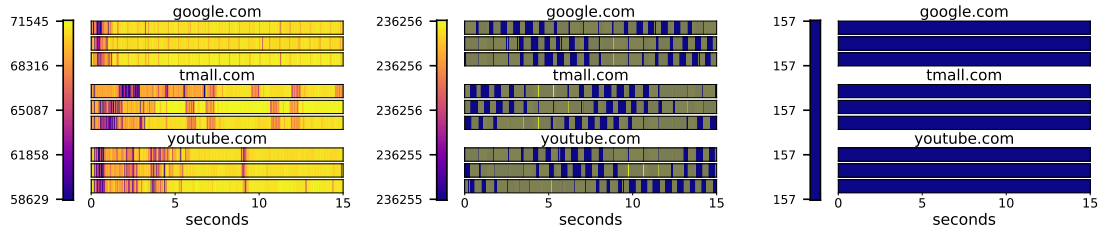
Modern processors strive to balance frequency, heat dissipation, and power consumption. A key mechanism for maintaining this balance is dynamic voltage and frequency scaling (DVFS), which adjusts the operating frequency and voltage. Recent studies have shown that frequency scaling can be exploited to leak information (Wang et al., 2022, 2023; Taneja et al., 2023).

To control leakage through DVFS, we perform the following procedure. First, we disable two BIOS features that allow the processor to change its frequency: TurboBoost and SpeedStep. Afterwards, we configure the operating system to fix the processor frequency at a level low enough for sustained operation. This is done using the `cpufreq-set` utility to set both the minimum and maximum allowed frequencies to the same value – half of the processor base frequency (1.8 GHz). Finally, we set the power governor to 'performance' and write a value of 0 to the `/dev/cpu_dma_latency` interface to prevent the processor from entering any low-power states.

Although many of these interfaces are provided on a per-core basis, consumer processors typically support only a single frequency domain, meaning the operating system must select one frequency for all cores. Since the processor frequency is set below its maximum, overall system performance is reduced. We account for this by ensuring that the time allotted to record each trace is sufficient for the browser to fully load the webpage despite the reduced performance.

3.2.4 Cache

To control leakage through the cache, we employ Intel Cache Allocation Technology (CAT) to partition the last-level cache. Specifically, we assign half of the cache ways to the measurement core and the other half to the remaining cores. This partitioning prevents other programs on the machine from evicting memory used by the measurement core, effectively severing cache eviction as a channel. However, as discussed below, this approach does not fully close all memory-based channels, and some leakage persists.



(A) Cache Occupancy
Machine: i7-7700
Accuracy: 91.9±2.0%

(B) Loop Counting
Machine: i7-7700
Accuracy: 11.3±2.6%

(C) MWAIT
Machine: i9-12900K
Accuracy: 10.0±0.0%

FIGURE 3.2: **Reduced Leakage:**
Measurements of the cache occupancy, loop counting, and mwait primitives after removing leakage.

3.2.5 Validating Control

Having implemented techniques to control each channel, we measure the effect of this control on each primitive. We apply all previous control measures and perform website fingerprinting as described in Section 3.1.1. Figure 3.2 illustrates the results.

Loop counting and mwait. We successfully eliminate all leakage measured by the loop-counting and mwait primitives. For the loop-counting primitive, the measurements change only minimally, and the variations do not appear correlated with browser activity. For the mwait primitive, all recorded samples are identical. This contrasts sharply with Figure 3.1, where leakage is so great that websites are distinguishable through visual inspection.

This elimination of leakage indicates that we have complete control over microarchitectural channels leaking information to the loop-counting and mwait primitives.

Cache Occupancy. Unfortunately, we do not have complete control over microarchitectural channels leaking information to the cache-occupancy primitive. Websites still remain distinguishable through visual inspection, as shown in Figure 3.2a. Moreover, training a classifier on these traces yields an accuracy of 91.9%.

This result already demonstrates a significant difference between the primitives. Specifically, that the primitives do not measure the same channels.

3.2.6 Remaining Leakage

We suspect that the uncontrolled channel is due to contention in off-core resources involved in servicing memory accesses. Several studies have shown leakage through inter-core interconnects (Dai et al., 2022; Paccagnella et al., 2021; Wan et al., 2022), last-level cache slices (Dai et al., 2022; Paccagnella et al., 2021; Wan et al., 2022), memory controllers (Wang et al., 2014), and DRAM (Pessl et al., 2016).

Experimental Description While fully controlling all these resources is beyond the scope of this thesis, we conduct the following experiment to reveal whether the leakage arises from these resources or not. We modify the cache-occupancy primitive so that its memory fits entirely within the private cache of the measurement core. Specifically, we reduce the size of the linked list to 28 KB, which is slightly smaller than the size of the L1 data cache.

Results. When training a classifier on recorded traces under this configuration, we achieve an accuracy of $11.8 \pm 2.6\%$. This is in stark contrast to the 91.9% accuracy when using a larger linked list, indicating that we have completely controlled the residual leakage. Since memory accesses under this configuration can be completely served without using off-core resources, we conclude that the leakage is due to contention in off-core resources. We leave the task of identifying and controlling these specific resources to future work.

3.3 Verifying Channel Contributions

Having established control over all channels, we now turn to the question: *Which channels leak information?*

To address this question, we begin with a system configuration from the previous section that completely controls leakage. For simplicity of analysis, we use the loop-counting primitive in this section.

For each channel, we create a system configuration that allows leakage exclusively through that channel. We create this configuration by applying all control measures from the previous section, except we omit those corresponding to the chosen channel. We then perform website fingerprinting on this new system configuration. If the classifier can successfully distinguish websites, we conclude that the channel leaks information.

Throughout this section, we find that all identified channels exhibit enough leakage to mount reliable website-fingerprinting attacks with the loop-counting primitive. Surprisingly, this includes the cache channel, despite the fact that the loop-counting primitive never performs any memory accesses.

3.3.1 Intracore Contention

First, we test whether intracore contention serves as a channel for information leakage. For this configuration, we omit the control steps described in Section 3.2.1 and manually relocate all interrupts away from the measurement core.

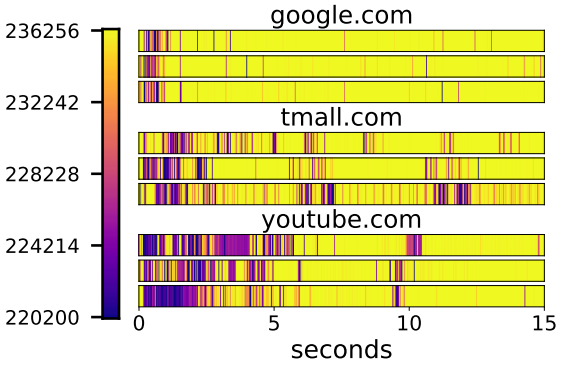


FIGURE 3.3: **Intracore Leakage:**
Traces recorded on i7-7700 with a configuration that allows leakage from intracore contention. Accuracy: 47.6±2.9%

Figure 3.3 illustrates the results of website fingerprinting under this configuration. The classifier achieves an accuracy of 47.6±2.9%. While this is notably lower than the accuracy obtained with the baseline configuration, it remains significantly higher than random guessing. We use this result, along with visual inspection of the heatmaps, to conclude that intracore contention leaks information.

3.3.2 Interrupt Handling

Next, we examine whether interrupt handling serves as a channel that leaks information. For this configuration, we omit the control steps described in Section 3.2.2.

In order to control intracore leakage, we use the `isolcpus` kernel parameter. However, this will cause the kernel to move interrupts away from the measurement core. To avoid falsely concluding that interrupts do not leak information, we manually relocate all interrupts back to the measurement core.

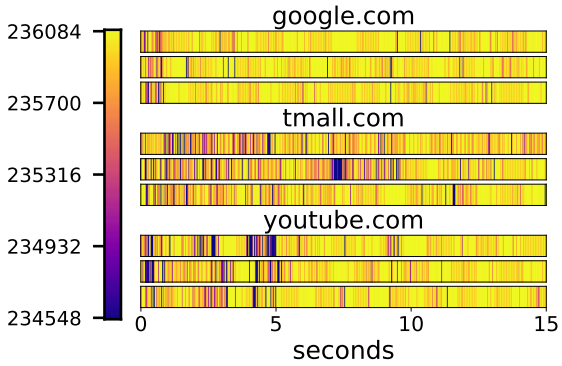


FIGURE 3.4: **Interrupt Leakage:**
Traces recorded on i7-7700 with a configuration that allows leakage from interrupts. Accuracy: 78.6±1.7%

When we perform the experiment with this configuration, the classifier achieves an accuracy of 78.6±1.7%. Figure 3.4 illustrates a heatmap of the resulting traces. Similar to the previous section, we use the classifier accuracy in combination with visual inspection to conclude that interrupt handling leaks information.

Per-device Interrupt Handling. The default configuration on our system distributes interrupt handling across all of the cores. For example, the operating system may assign one core to handle all interrupt requests from the network interface, while another core handles all interrupts from the sound interface. This raises a natural question: *Do different interrupts leak different information?*

Experiment design. We begin with a system configuration that completely removes leakage using the steps described in the previous section. Afterwards, we move all interrupts from a specific device to the measurement core while relocating all other interrupts to a different arbitrarily chosen core. We then perform website fingerprinting as outlined in Section 3.1.1. This experiment is repeated for each device in the system: the USB controller (USB), NVMe storage (Storage), networking, graphics, sound, and the system management engine (Management).

TABLE 3.2: **Interrupt Leakage (By device):**
Accuracy of the loop counting primitive on our i7-7700 machine when we deliver interrupts from a specific device to the measurement core.

Interrupt	Accuracy
Graphics	$61.56 \pm 3.01\%$
Management	$11.24 \pm 1.70\%$
Networking	$56.20 \pm 2.25\%$
Sound	$22.36 \pm 2.34\%$
Storage	$54.92 \pm 2.09\%$
USB	$10.60 \pm 2.08\%$

Results. Table 3.2 shows the results of this experiment. The classifier accuracy varies significantly depending on which device interrupts are delivered to the measurement core. For example, interrupts from the internal graphics card yield the highest accuracy at 61.6%, while interrupts from the management engine produce the second-lowest accuracy at 11.2%. The lowest accuracy is seen with USB interrupts; however, this outcome is likely an artefact of our experimental setup since the device under test is exclusively used remotely and has no USB peripherals attached. Using the results of this experiment, we conclude that different interrupts do leak different information.

3.3.3 Frequency Scaling

Next, we examine whether frequency changes serve as a channel to leak information. For this configuration, we omit the control steps described in Section 3.2.3.

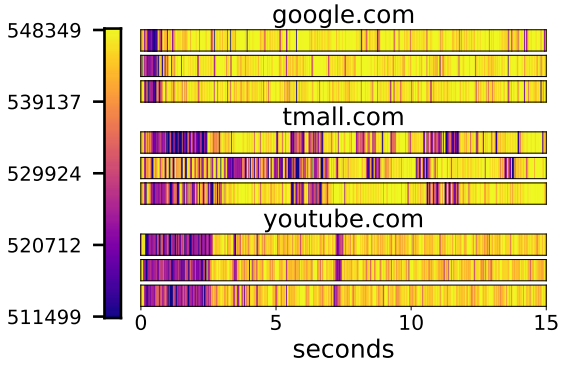


FIGURE 3.5: **Frequency Scaling Leakage:**
Traces recorded on i7-7700 with a configuration that allows leakage from frequency changes. Accuracy: 90.8±1.5%

Results. Figure 3.5 illustrates heatmaps from this experiment. When we train a classifier on traces recorded under this configuration, it achieves an accuracy of 90.8±1.5%. Again, we use these results to conclude that frequency scaling due to DVFS leaks information.

3.3.4 Cache

Finally, we investigate the cache as a channel for leaking information. For this final configuration, we omit the steps from Section 3.2.4.

Results. As with all other configurations, the traces illustrated in Figure 3.6 can be distinguished by eye. Furthermore, the classifier achieves an accuracy of 94.3±1.2%. At first glance, this result is not too surprising due to existence of many works that explicitly use the cache to leak information; however, the loop-counting primitive that we use to collect the measurements does not perform any memory accesses.

This raises a natural question: *How does loop counting measure cache leakage?* The remainder of this section addresses this question.

Measuring Cache Misses. We begin by instrumenting the loop-counting primitive to record the number of cache misses in the L1I (instruction) and L1D (data) caches. We

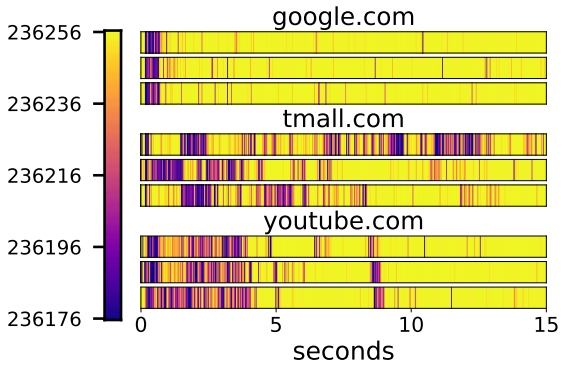


FIGURE 3.6: **Cache Leakage:**
Traces recorded on i7-7700 with a configuration that allows leakage from cache evictions. Accuracy: 94.3±1.2%

achieve this by programming the hardware performance counters to track the number of `FRONTEND_RETIRED.L1I_MISS_PS` and `MEM_LOAD_RETIRED.L1_MISS_PS` events during each measurement. When we perform the experiment again, we observe no data cache misses, but we do observe instruction cache misses.

Root Cause Analysis. We hypothesise that the cause of these instruction cache misses lies in the architecture of the inclusive last-level cache (LLC) of the i7-7700 processor. In an inclusive cache hierarchy, when a cache line is evicted from the LLC, it is also invalidated from the private caches (L1I, L1D, and L2) of each core. Since the LLC does not differentiate between data and instructions, extensive data accesses by the victim may inadvertently evict the loop-counting primitive's code from the cache. When such eviction occurs, the processor loses immediate access to those instructions and stalls while it fetches them from system memory. This stall reduces the number of loop iterations completed during the measurement interval, creating a signal that is correlated with the victim's memory activity.

Non-Inclusive Caches. To test this hypothesis, we repeat the experiment with a i9-12900KF processor, which features a non-inclusive LLC. In such architectures, evicting a cache line from the LLC does not cause it to be evicted from the private caches of each

core.

When we conduct the experiment, we observe no L1I cache misses, and the loop-counting primitive loses all sensitivity to cache leakage. These results support our hypothesis: the loop-counting primitive is sensitive to cache leakage on inclusive-cache systems because its instructions are being evicted from the instruction cache as a result of victim memory activity.

3.4 Measuring Channel Contributions

In the previous sections, we demonstrate our ability to control leakage (Section 3.2) and show that all channels leak information (Section 3.3). This section measures the relative contribution of each channel to the total leakage measured by each primitive.

3.4.1 Methodology

We perform website fingerprinting while simultaneously recording various system and processor performance events related to each channel. We then compute the Pearson correlation coefficient between the number of iterations and the collected performance events, using this coefficient as an indicator of each channel's contribution to the total leakage measured by each primitive. We report the average and standard deviation of the coefficient across all 1,000 traces.

We modify the inner loop of each primitive to track various processor events. For each 5-millisecond sample, we record the number of loop iterations, the number of interrupts delivered to the measurement core, the total time spent handling interrupts, the processor frequency, and the number of L1I and L3 cache misses attributed to the primitive. All these measurements are performed on a baseline system, meaning we do not apply any of the controls described in Section 3.2.

Measuring Interrupts. As in previous sections, we measure the number of interrupts using the `/proc/interrupts` and `/proc/softirqs` interface. Sampling occurs every 5 milliseconds, and we compute the difference between successive samples.

Measuring Interrupt Time. While the total number of interrupts can be measured easily, there are no readily available interfaces to capture the total interrupt handling

time with fine granularity. To address this, we adopt the same technique used by Cook et al. (2022), which detects gaps in execution and attributes them to interrupt handling. Specifically, we modify the inner loop to repeatedly access the `RDTSCP` instruction. If the difference between successive measurements exceeds 500 nanoseconds, the interval is recorded as an execution gap, and its duration is added to the total interrupt handling time.

Although this thesis attributes all execution gaps to interrupt handling time, our currently unpublished follow-up work reveals that many of these gaps are caused by other factors, such as the processor pausing execution to adjust its frequency.

Measuring Frequency. We measure the `CPU_CLK_UNHALTED.THREAD_ANY` and `CPU_CLK_UNHALTED.REF_TSC` performance events. These events serve functions similar to the `APERF` and `MPERF` model-specific registers but can be accessed directly from user space. The ratio between these two events provides the average processor frequency over a given period (Intel, Vol. 3B §14.2).

Measuring Cache Activity. We measure data and instruction cache activity separately by monitoring the `CYCLE_ACTIVITY_CYCLES_L3_MISS` and `FRONTEND_RETIRED.L1I_MISS` performance events. The former counts the number of cycles during which load instructions are stalled due to L3 cache misses, while the latter counts the number of instructions that miss in the L1I cache.

3.4.2 Contributions of Channels

Here, we report the correlation between each channel and the iteration count for each primitive. Since the operating system assigns specific cores to handle interrupts from specific devices, the results may vary depending on the devices connected to the system and the core used for measurements. To account for this, we perform the experiments under two system configurations: a baseline configuration where we arbitrary selected the final core on the system as the measurement core and a modified configuration where all interrupts are redirected to the measurement core.

Table 3.3 presents the results for all three primitives under these two configurations. We restrict our reporting to one column of the table at a time – the column corresponding to the primitive of interest in each of the following paragraphs.

TABLE 3.3: **Correlating Channels and Measurement Primitives:** Correlation between iterations and number of events on both machines. Baseline refers to the default interrupt configuration. Interrupts refers to the configuration where all interrupts are delivered to the measurement core.

		Cache occupancy		Loop counting		mwait
	Machine	i9-12900KF	i7-7700	i9-12900KF	i7-7700	i9-12900KF
Baseline	Interrupt Count	0.04 ± 0.01	-0.03 ± 0.02	-0.05 ± 0.03	0.03 ± 0.05	0.11 ± 0.04
	Interrupt Time	-0.20 ± 0.07	-0.76 ± 0.03	-0.27 ± 0.07	-0.40 ± 0.05	$\mathbf{0.79 \pm 0.08}$
	Frequency	0.70 ± 0.04	0.50 ± 0.11	$\mathbf{0.77 \pm 0.09}$	$\mathbf{0.82 \pm 0.04}$	0.48 ± 0.19
	L1I Cache	-0.15 ± 0.03	0.08 ± 0.05	0.04 ± 0.02	0.06 ± 0.04	-0.06 ± 0.17
	L3 Cache	$\mathbf{-0.99 \pm 0.00}$	$\mathbf{-0.93 \pm 0.04}$	-0.10 ± 0.04	-0.44 ± 0.06	-0.09 ± 0.04
Interrupts	Interrupt Count	-0.25 ± 0.05	-0.17 ± 0.05	-0.23 ± 0.05	-0.19 ± 0.04	$\mathbf{0.55 \pm 0.15}$
	Interrupt Time	-0.36 ± 0.06	-0.76 ± 0.02	-0.35 ± 0.07	-0.44 ± 0.06	0.44 ± 0.13
	Frequency	0.70 ± 0.03	0.49 ± 0.08	$\mathbf{0.78 \pm 0.08}$	$\mathbf{0.81 \pm 0.04}$	0.22 ± 0.18
	L1I Cache	-0.22 ± 0.05	-0.28 ± 0.04	-0.20 ± 0.05	-0.22 ± 0.05	0.40 ± 0.12
	L3 Cache	$\mathbf{-0.98 \pm 0.00}$	$\mathbf{-0.93 \pm 0.03}$	-0.22 ± 0.06	-0.45 ± 0.05	0.15 ± 0.08

Cache occupancy. For the cache-occupancy primitive, L3 cache misses shows the highest correlation with the number of iterations. This is irrespective of whether the processor features an inclusive (i7-7700) or non-inclusive (i9-12900KF) cache or whether interrupts are moved to the measurement core or not.

Loop counting. For the loop-counting primitive, frequency shows the highest correlation with the number of iterations. Interestingly, interrupt count and interrupt time exhibit significantly different correlations to the number of iterations. Our unpublished follow-up work investigates this result further.

mwait. Finally, we report the results for the mwait primitive. As this primitive relies on instructions supported only by newer architectures, we limit our analysis to the i9-12900KF processor. Under the baseline configuration, interrupt time shows the highest correlation. When interrupts are redirected to the measurement core, interrupt count shows the highest correlation.

Comparing to Cook et al (2022). Cook et al. (2022) suggest that the primary source of leakage for the cache-occupancy and loop-counting primitives is interrupt-related activity. In contrast, our findings indicate that the main contributors are cache behaviour and

frequency scaling, respectively. This discrepancy likely arises from differences in how channel contribution is measured.

Using machine learning accuracy as a metric can be misleading, as accuracy may improve even when information is removed from the signal. Consider loop-counting measurements on a system with a baseline configuration (Figure 3.1b) and a modified configuration that only allows leakage through the cache channel (Figure 3.6). Under the first configuration, we trained a classifier with an accuracy of $93.4 \pm 0.8\%$. Under the second configuration, we trained a classifier with a higher accuracy ($94.3 \pm 1.2\%$) despite the removal of channels that contain information.

We hypothesise that this increase in accuracy occurs because leakage from different channels may destructively interfere with each other, thereby reducing overall signal quality. In contrast, using Pearson correlation as a measure of channel contribution accounts for such interference, leading to different insights than those obtained from classifier-based analysis alone.

3.5 Conclusion

Throughout this chapter, we describe how to control leakage through microarchitectural channels and how to use this control to uncover insights into how leakage propagates throughout a system. We use this approach to analyse recent microarchitectural website-fingerprinting attacks and address conflicting explanations surrounding the cache-occupancy attack.

Our findings show that while the cache emerges as the dominant channel for the cache-occupancy attack, other channels also contribute significantly. We observe similar patterns in our analysis of the loop-counting and mwait primitives too.

Our findings highlight the importance of considering multiple channels when attempting to mitigate the risk of microarchitectural attacks. Countermeasures that only address the most prominent channel may not be sufficient to eliminate the threat of attack. More concerning is the impracticality of isolating all potential channels, particularly those involving the memory hierarchy, suggesting that fully mitigating cache-occupancy or loop-counting attacks may not be feasible.

Fortunately, sensitivity to multiple channels seems limited to coarse-grained attacks such as the website finger-printing attacks analysed in this chapter. These attacks seem to

have limited ability to extract sensitive information, reducing the urgency for immediate and effective countermeasures.

The following chapters explore significantly more powerful adversaries capable of conducting higher-bandwidth measurements and examine the implications of these capabilities on the security of modern web browsers.

Chapter 4

Mounting a High-Capacity Pixel-Stealing Attack

When web resources interact, the browser enforces a security policy to determine whether the interaction should be allowed. The most common policy is the same-origin policy, which restricts interactions between resources served from different origins.

However, web browsers do permit certain forms of cross-origin interaction. One such example is the cross-origin application of SVG filters – small functions that alter the visual appearance of web content and enable effects, such as blurring, that are otherwise difficult to achieve using standard HTML and CSS alone. Web browsers allow filters to be applied to cross-origin resources to create cohesive visual effects on webpages. This is assumed to be safe because the browser restricts access to the output of the filter.

Unfortunately, unintended side effects from filter application may be observable to malicious websites. For example, previous studies have shown that filter execution time is data dependent, enabling attackers to exploit these timing variations to launch pixel-stealing attacks (Stone, 2013; Andrysco et al., 2015; Kotcher et al., 2013).

Figure 4.1 illustrates the general structure of such filter-based pixel-stealing attacks. In these attacks, the attacker lures a user to a malicious website that embeds sensitive content, such as a third-party site within an `iframe`, and then applies a filter to the displayed content. The content influences the execution time of the filter which, in turn, influences the page rendering time. While the attacker cannot directly measure the page rendering time, they can induce a large delay to cause the browser to miss the deadline to render the next frame – an event the attacker can detect.

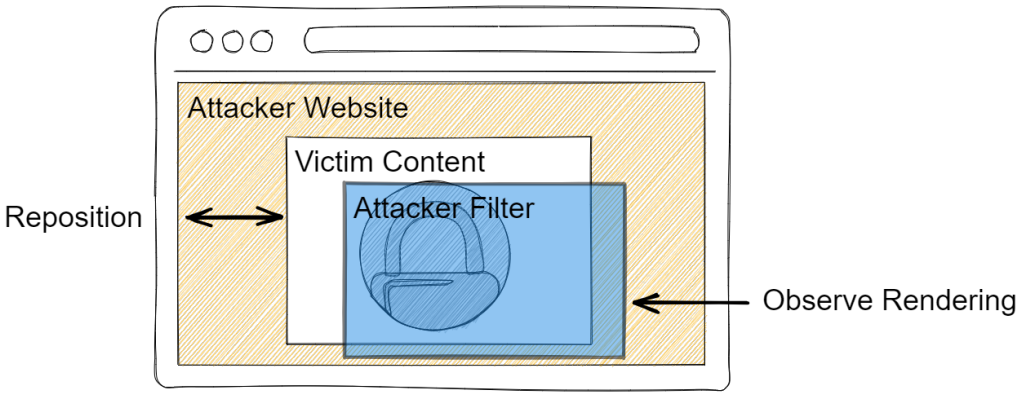


FIGURE 4.1: **Overview of a Pixel-Stealing Attack:** An adversary tricks a user into visiting their malicious website (*Attacker Website*). The Attacker Website embeds a website that displays sensitive user information (*Victim Content*) then applies a vulnerable SVG filter to the content (*Attacker Filter*). The adversary observes how the filter impacts the rendering of the browser to extract information and repositions the victim content to extract the desired subset of the victim content.

A fundamental limitation of these pixel-stealing attacks is that they extract information by causing the browser to miss the deadline for rendering a frame. Considering that browsers render frames at a fixed rate, typically matching the refresh rate of the display, the data rate of such attacks is inherently limited. This limitation is further compounded by the fact that attacks usually extract only a single binary value per frame, which typically restricts the maximum data rate to no more than 60 bits per second. Furthermore, browser vendors have acknowledged the risk of pixel-stealing attacks and implemented countermeasures to eliminate observable data-dependent timing differences in filter execution (Mozilla Bug Tracker, 2013; Chromium Project, 2013, 2016a, 2017). Recent studies have shown that frequency scaling can be exploited to mount pixel-stealing attacks despite these mitigations (Taneja et al., 2023; Wang et al., 2023), but these attacks suffer from even lower leakage rates. This raises the question: *Are high-capacity pixel-stealing attacks on modern browsers feasible?*

This chapter answers the question in the affirmative by presenting a high-capacity pixel-stealing attack capable of leaking several hundred bits per second. The attack achieves this by employing a different mechanism to extract information from the browser: it uses the Prime+Probe technique to monitor content-dependent memory accesses performed by the `feComponentTransfer` SVG filter. Since the filter performs memory access for each pixel, the increased number of measurable events raises the data rate well beyond the 60-bit-per-second limit.

Section 4.1 describes a generic technique to bypass cross-origin isolation, a countermeasure designed to prevent malicious websites from exploiting microarchitectural event measurements. Sections 4.2 and 4.3 introduce Pixel Thief, a cache-based pixel-stealing attack that exploits data-dependent memory accesses in the `feComponentTransfer` filter. Finally, Sections 4.4 and 4.5 demonstrate how Pixel Thief can be used as a primitive for efficient text-stealing and history-sniffing attacks, respectively.

4.1 Overcoming Cross-Origin Isolation

The first major challenge for pixel-stealing attacks on modern browsers is cross-origin isolation. To access high-resolution timers directly or to create them using `SharedArrayBuffer`, a website must control cross-origin isolation with the *COEP/COOP* headers. If the website embeds or is embedded within a cross-origin site, both sites must use these headers and must include each other on their embedding allow-list.

While it may be possible to exploit incorrectly configured websites, the attacker is limited to targetting only these misconfigured websites. To allow the attack to target the vast majority of sites, we focus on scenarios where these headers are not used.

In such scenarios, the browser prevents access to high-resolution timers, which limits the attacker's ability to mount high-capacity attacks. One potential solution is to build on prior work by employing advanced techniques that leverage transient execution to enable side-channel attacks with high-temporal accuracy even without a source of high-resolution timing (Katzman et al., 2023; Kaplan, 2023). However, before adopting these complex methods, we ask the question: *Does the mutual exclusion between embedding websites and accessing high-resolution timers actually exist?*

The rest of this section demonstrates that the answer is no, provided the attacker is willing to accept stricter constraints on their attack. Specifically, the adversary must

coerce the victim into interacting with the malicious website at least once. This shifts the adversarial model from those used in previous pixel-stealing works to the slightly stricter models used in *clickjacking* attacks (Huang et al., 2012).

Bypassing cross-origin isolation. We use this interaction to open a second page. The first page sends *COEP/COOP* headers to access high-resolution timers, but cannot embed cross-origin content. The second page is reversed, it cannot access high-resolution timers because it does send *COEP/COOP* headers, but it can embed cross-origin content.

The attack is split between these two pages: any component requiring high-resolution timers runs on the first page, while any part interacting with cross-origin content runs on the second page. The two pages cannot communicate directly owing to cross-origin isolation. However, both can send messages to a server via `WebSockets`, which can relay messages between the two pages.

Figure 4.2 illustrates an overview of this approach. The server serves two pages to the victim: the first page disables *COEP/COOP* and embeds the victim page (blue webpage on the left), while the second page enables *COEP/COOP* and performs the cache attack with high-resolution timers (orange webpage on the right). (#1) The server instructs the first page to manipulate the victim page to leak data. (#2) The data leaks through the cache from the victim to the second page. (#3) The second page recovers the leaked data using the high resolution timers and sends it back to the server.

Same-Origin Content. When the targetting same-origin content, such as the history-sniffing attack described in Section 4.5, the two-page architecture is unnecessary because there is no cross-origin content that the browser restrict access to. We use the single-page architecture for attacks that leak same-origin content and the two-page architecture for those that leak cross-origin content.

4.2 Leaking Pixels

This section describes the pixel-stealing attack that serves as the foundation for the other attacks presented in this chapter. The pixel-stealing attack reveals sensitive data displayed in rendered webpages by exploiting data-dependent memory accesses in filters. Subsequent attacks in this chapter encode secret information into parts of the rendered webpage and then use this attack to recover it.

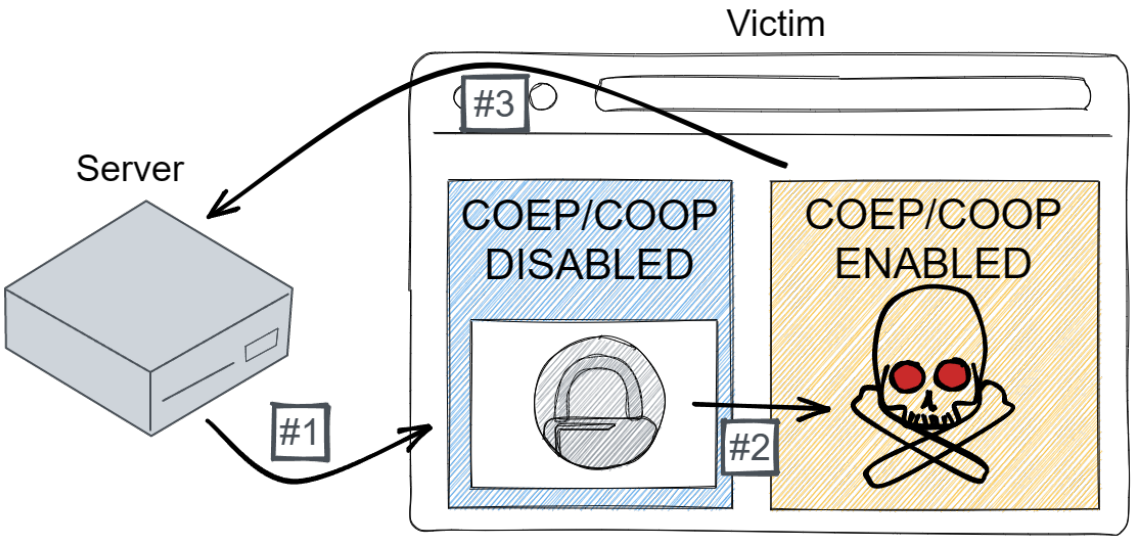


FIGURE 4.2: **Bypassing Cross-Origin Isolation:**
The server split the attack into two pages that can be delivered to the user under different security models. The first page embeds sensitive content (blue/left) while the second mounts the cache attack (orange/right). (#1) The server instructs the first page to leak information (#2) which is recovered via a cache attack and then (#3) transmitted back to the server.

Figure 4.3 illustrates an overview of the attack. The attacker embeds the victim image into their page (#1) and uses CSS to isolate and scale specific pixels (#2). Then, the attacker prepends their own image above these pixels (#3), so that when the filter is applied, it generates a predetermined sequence of memory accesses. When the browser renders the page, the filter processes both images using each pixel value as an offset into a lookup table (#4). In parallel, the attacker performs a cache attack to monitor memory accesses to this table over time (#5). Finally, the attacker analyses the recorded memory accesses to identify the predetermined sequence corresponding to their own image. Subsequent memory accesses are attributed to the victim's image and can be recovered then combined with previously obtained pixels to reconstruct the entire victim image (#6).

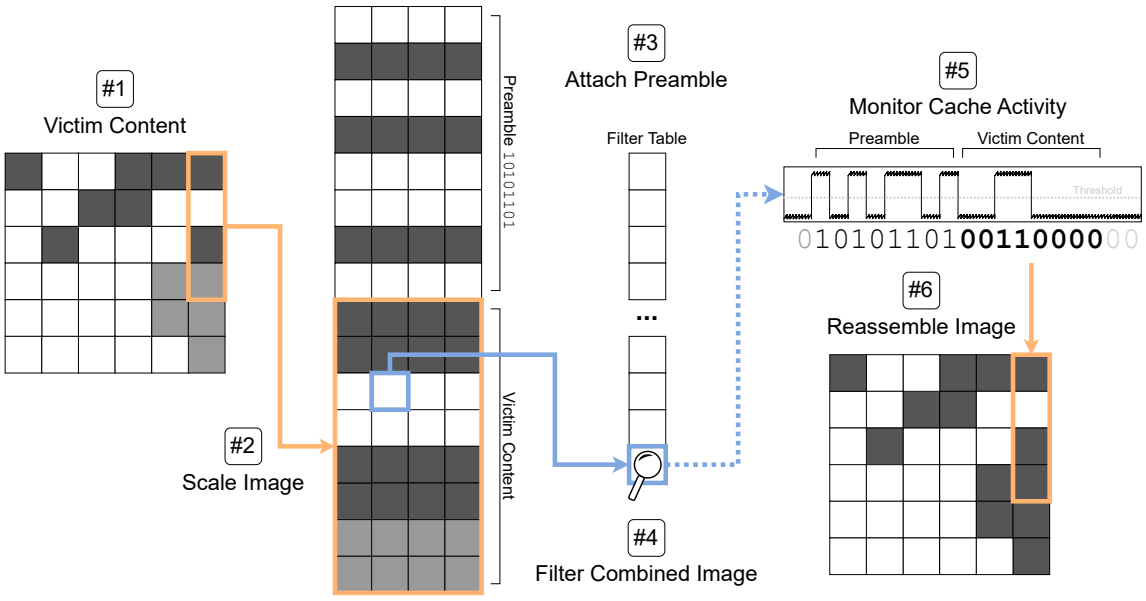


FIGURE 4.3: **Recovering Pixel Data:**
(#1) Embed victim content. (#2) Isolate pixels in victim content. (#3) Embed attacker content. (#4) Apply filter to both images. (#5) Record memory accesses. (#6) Find memory accesses correlating to attacker content then recover victim content.

While the attack works in both Chrome and Firefox, Firefox's implementation of the relevant filter is significantly easier to explain; therefore, the remainder of this section focuses on Firefox. It begins with a description of the vulnerable SVG filter and then discusses GPU implementations of SVG filters, the challenges these pose for the attacker, and how the attacker can force Firefox to use the vulnerable CPU implementations instead. Finally, the section concludes with a discussion on measurement frequency.

4.2.1 The feComponentTransfer Filter

The `feComponentTransfer` SVG filter element enables designers to remap the colours of their webpages. It can be used to create various recolouring effects, such as sepia

or grayscale. The filter processes each pixel of the input image individually: first, it separates the pixel into its red, green, and blue components; then, it applies designer-defined functions to each component's value independently; finally, it recombines the components to produce the pixel colour in the output image. Listing 4.1 (Lines 2–6) provides an example of how such a filter is defined.

```
1 <filter id="filter_id">
2   <feComponentTransfer>
3     <feFuncR type="discrete" tableValues="0 1"></feFuncR>
4     <feFuncG type="discrete" tableValues="1 0"></feFuncG>
5     <feFuncB type="discrete" tableValues="0.3 0.6 0.9"></feFuncB>
6   </feComponentTransfer>
7   <feGaussianBlur stdDeviation="0" />
8 </filter>
```

Listing 4.1: **Malicious filter definition:**
A filter that executes the vulenrable `feComponentTransfer` filter followed by feGaussianBlur to force execution of the filter onto the CPU.

Implementation. The `feComponentTransfer` specification allows several ways to define the colour mapping functions, including identity maps, linear maps, gamma maps, interpolated tables, and discrete tables (MDN Contributors, a). Regardless of the chosen method to define the colour mapping functions, Firefox implements the function using a 256-entry lookup table that is populated during a precomputation phase.

Listing 4.2 shows a simplified version of Firefox's `feComponentTransfer` implementation in c-like pseudocode. Lines 1–4 define the implementation function and its parameters. The `input` and `output` parameters represent the filter's input and output images, each composed of interleaved 8-bit red, green, and blue values. The final parameter, `tables`, contains the precomputed lookup tables generated from the colour mapping functions defined for the filter. Line 6 iterates over every pixel in both the input and output images, while Line 7 iterates over each colour component of the current pixel. Finally, Line 8 applies the filter by performing a lookup in the corresponding table.

```
1  void TransferComponents(
2    uint8_t input[N],
3    uint8_t output[N],
4    uint8_t tables[3][256]
5  ) {
6    for (uint32_t i = 0; i < N; i += 3) {
7      for (uint32_t c = 0; c < 3; c++) {
8        output[i + c] = tables[c][input[i + c]];
9      }
10   }
11 }
```

LISTING 4.2: **Firefox's feComponentTransfer implementation.**

Memory access patterns. Here, we analyse the cache access patterns of the `feComponentTransfer` implementation. Consider an image with two pixels: the first is black $(0, 0, 0)$, and the second is white $(255, 255, 255)$. As before, Lines 6 and 7 of Listing 4.2 iterate over each component of each pixel, with Line 8 performing table lookups. A side effect of these lookups is that the accessed table entries are loaded into the CPU cache hierarchy. Specifically, applying the filter to the black pixel loads the first entries of each table into the cache, while processing the white pixel loads the last entries of each table. Consequently, an attacker who can distinguish between memory accesses to the first entries versus the last can determine whether the pixel is black or white.

4.2.2 Executing feComponentTransfer on the CPU

To improve rendering performance, Firefox version 92.0 and later attempts to offload image rendering to the GPU. This GPU-based rendering creates a challenge for adversaries seeking to exploit vulnerabilities in the CPU implementations of filter elements, as these vulnerable implementations are bypassed. We describe how to overcome this issue by forcing Firefox to execute filter elements on the CPU instead.

TABLE 4.1: **Filter Execution Location:**
List of filter elements. Filter elements marked ✓ have a GPU implementation and elements marked ✗ do not. feImage and feMerge use a separate system that is out of scope for this work and are marked —.

Filter Element	GPU Support
feBlend	✓
feColorMatrix	✓
feComponentTransfer	✓
feComposite	✓
feConvolveMatrix	✗
feDiffuseLighting	✗
feDisplacementMap	✗
feFlood	✓
feGaussianBlur	✗
feImage	—
feMerge	—
feMorphology	✗
feOffset	✓
feSpecularLighting	✗
feTile	✗
feTurbulence	✗

Filter Elements without GPU Implementations. Although Firefox offers GPU-based implementations for many filter elements, some filter elements are still supported only via CPU-based implementations; see Table 4.1 for a complete list. One example is `feGaussianBlur`, which applies a Gaussian blur (Waltz and Miller, 1998) to an image. However, none of these CPU-only filter elements exhibit data-dependent memory access patterns capable of leaking image data into the cache.

Forcing CPU Computation. Unable to construct an attack using filters with only CPU-based implementations, we explore methods to force Firefox to fall back to executing `feComponentTransfer` on the CPU. To this end, we examine Firefox's behaviour when

combining filter elements, particularly mixes of CPU- and GPU-based filters.

We find that when `feComponentTransfer` is combined with `feGaussianBlur`, both filters execute on the CPU. [1] Moreover, we find that this pattern holds consistently: if the last filter in the stack is CPU-based, the entire filter runs on the CPU, regardless of how many GPU-based filters are included. [2]

Our Filter. To exploit this behaviour, we use the filter shown in Listing 4.1. It consists of an `feComponentTransfer` filter element (Lines 2–6), whose output is fed into an `feGaussianBlur` filter element (Line 7). Since Firefox lacks a GPU implementation for `feGaussianBlur`, it falls back to the CPU implementation for that filter. Consequently, `feComponentTransfer` is also executed using its CPU-based implementation as shown in Listing 4.1. The choice of `feGaussianBlur` is arbitrary – any other CPU-only filter could be used to force filter execution on the CPU.

Chrome. Chrome exhibits behaviour similar to Firefox, preferring to execute `feComponentTransfer` on the GPU. While we could not identify a method to force Chrome to run `feComponentTransfer` on the CPU, it does maintain an extensive GPU blocklist that disables various GPU acceleration features on specific hardware configurations. These configurations include devices with outdated drivers, systems running Windows Vista or earlier, Linux machines with third-party drivers, and MacOS devices without a GPU on the allowlist (Chromium Project, 2023). Consequently, an attacker would be limited to targeting victims using devices in one of these restricted configurations. In our experiments, we simulate this scenario by launching Chrome with the `--disable-gpu` flag, which disables GPU support entirely.

Signal Amplification. Cache attacks have a limited measurement frequency, which restricts their ability to distinguish between rapidly occurring events (Allan et al., 2016; Purnal et al., 2021). Our goal is to recover an image by measuring filter memory accesses with a cache attack. However, the filter operates too quickly – it performs memory accesses every few CPU cycles, while a typical cache attack requires hundereds or even

[1] We note that if the filters are combined in the reverse order `feComponentTransfer` is executed on the GPU.

[2] We note that explicit linking of filter-elements using `in` attributes interferes with this behaviour and prevents GPU-based filters from executing on the CPU.

thousands of cycles per measurement (Allan et al., 2016; Yarom and Falkner, 2014; Liu et al., 2015; Purnal et al., 2021).

Previous works have explored methods to reduce the execution speed of the transmitter (Allan et al., 2016; Aldaya and Brumley, 2022; Gullasch et al., 2011; van Bulck et al., 2017; Moghimi et al., 2017), but these techniques rely on features unavailable to JavaScript, so they cannot be used in the attack. Instead, we build on prior research (Andrysco et al., 2015; Kotcher et al., 2013; Stone, 2013) that stretches the image to create copies of each pixel forcing the filter to perform repeated memory accesses. Since Firefox breaks the screen into 256×256 px tiles, we stretch each pixel horizontally by 256 times and vertically by an amount that depends on the number of pixels recovered simultaneously.

4.3 Recovering Pixels

In the previous section, we construct the transmitter side of our pixel-stealing attack. This section shifts focus to the receiver side.

First, we explain how to prepare for mounting a cache attack and then describe the specific cache attack we employ. Our goal is to build a fast attack that enables a high data transfer rate. To achieve this goal, we use Prime+Probe (Osvik et al., 2006; Liu et al., 2015; Percival, 2005). Then we explain how we synchronise the transmitter and receiver, followed by an evaluation of the pixel recovery rate, and concluding with a comparison to existing works.

4.3.1 Detecting Transmitter Communications

Unlike previous filter-based attacks (Andrysco et al., 2015; Kotcher et al., 2013; Stone, 2013), our approach measures leakage concurrently with filter execution. However, the attacker cannot immediately determine whether a given cache measurement corresponds to memory activity of a filter or of some unrelated process running on the machine.

In this section, we explain how to distinguish cache measurements that represent transmitter communications from noise generated by other processes. We then show how this ability to detect transmitter activity enables the establishment of a reliable communication channel. We begin by assuming the attacker knows which cache sets to monitor and subsequently describe how these sets can be identified.

Adding the preamble. To detect the start of a transmission, we employ a standard communication technique by introducing a packet preamble. The preamble serves two purposes: first, it enables the receiver to identify the beginning of the transmission; second because its content is known, it allows the receiver to synchronise its clock rate (which is based on a free-running counting thread) to the rate of the sender. Recall that the signal is encoded as cache activity by applying an attacker-controlled filter over a page containing victim content. We insert the preamble by prepending a known image to the page, positioned just before the victim content so that the filter processes the preamble first, followed by the sensitive information.

```
1 <div style="filter: url(#filter_id);">
2   <img src="preamble.png" />
3   <div>
4     <!-- sensitive content -->
5   </div>
6 </div>
```

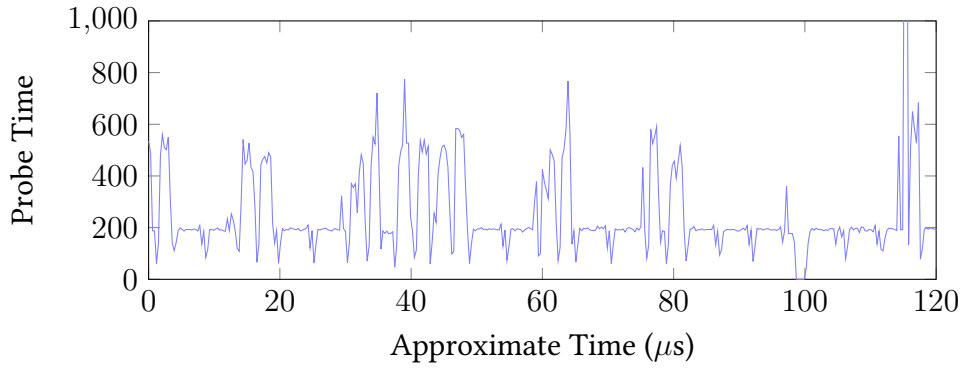
LISTING 4.3: **Applying a preamble:**
The filter is applied to page elements in their rendering order. We add an image before the sensitive content so that when the filter is applied to both, it first produces a known memory access pattern that can be detected.

Listing 4.3 illustrates how the preamble is added to the attacker webpage. Line 1 applies the filter to both the preamble and the sensitive content. Line 2 displays the preamble image on the screen, while Lines 3–5 display the sensitive content. Because the filter processes the image first horizontally, then vertically, it will first operate on the preamble before processing the sensitive content.

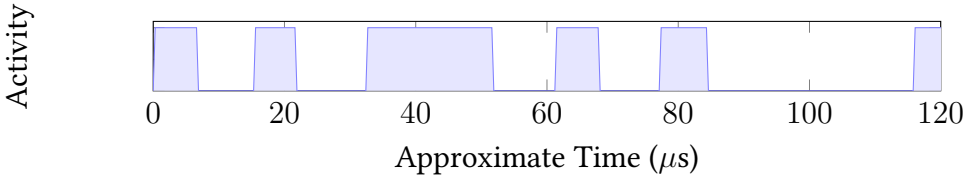
Detecting the preamble. With the transmitter set to send a preamble, we now focus on detecting it at the receiver. Because the exact rate of the receiver's counting thread is unknown to the attacker and may vary over time, it is not possible to directly search the side-channel trace for the preamble, as the signal may be stretched in time by an unknown factor. Instead, we search for the preamble across multiple potential stretching factors. To speed up this process, both the preamble and the trace are first processed using

run-length encoding. In this representation, time stretching corresponds to multiplying by a constant factor. Once the preamble is detected, the attacker gains knowledge of both the frame start time and the correct data rate for sampling the transmitted data.

We implement the detector as follows: the input signal first passes through a low-pass filter to remove high-frequency noise. Afterwards, the filtered samples are compared against a threshold to classify them as cache hits or cache misses. The resulting trace is then run-length encoded, and finally, a substring search is performed to locate the preamble. Figure 4.4 illustrates an example of cache activity measured by our attack, both before and after filtering the signal.



(A) Raw Data



(B) After Filtering

FIGURE 4.4: **Preamble Memory Access Patterns:** Memory access patterns of an SVG filter when rendering a preamble image that encodes the sequence `101011101010001`. Recorded using Prime+Probe then filtered with a low-pass filter and thresholded with a value of 300.

Packet Payload. Our attack can be configured to leak varying numbers of pixels simultaneously, depending on the trade-off chosen between speed and error rate. Each filter invocation effectively transmits a 'packet' of side-channel data, beginning with the preamble followed by the sensitive payload.

Identifying The Target Set. Until now, we have assumed that the attacker already knows which cache set is used by the filter processing code and only needs to detect when data transmission occurs. To identify the cache set used by the filter, the attacker first constructs an eviction set for each cache set using the technique described by Vila et al. (2019).

After constructing all eviction sets, the attacker repeatedly applies the filter to the preamble. Then, for each eviction set, the attacker uses Prime+Probe with the eviction set to measure memory activity on the system. The attacker measures a trace long enough to include one invocation of the filter then searches the recorded trace for the preamble. Upon detecting the preamble, the attacker records the corresponding eviction set, eliminating the need for this calibration in future measurements. If the preamble is not detected, the attacker moves on to the next eviction set.

4.3.2 Evaluation

In the previous section, we describe the final steps for detecting cache leakage from `feComponentTransfer`. In this section, we evaluate how varying the payload size, the time needed to identify the target cache set, and the presence of noise affect the leakage rate and accuracy of the attack.

Throughout this section, we mount our pixel-stealing attack on the pixel-art Firefox logo shown in Figure 4.5 (first image on the left). We start by evaluating the speed and accuracy trade-off when selecting payload size. Then we evaluate the capability of the attack to find the signal in the cache. Finally, we evaluate the attack under more realistic conditions, including an additional concurrent workload.

We use Prime+Probe to measure the final entry of the `feComponentTransfer` filter table, corresponding to indexes 192–256, and ignore accesses to any other indexes. For each pixel, if access to this cache line is observed, we assume that index 256 of the table was accessed; otherwise, we assume index 0 was accessed. Under these assumptions,

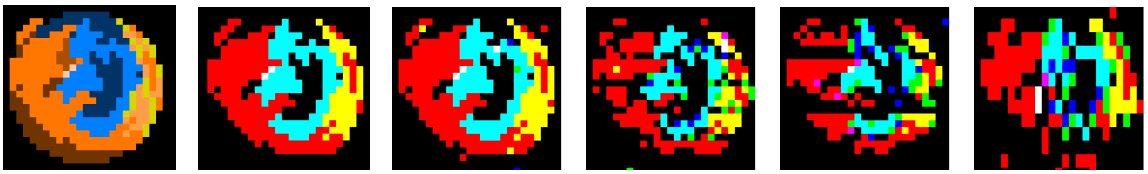


FIGURE 4.5: **Recovered Images:**
Pixel-art Firefox logos were recovered using our pixel-stealing attack. From left to right: original image, ideal leakage, and actual leakage with payloads of 1, 4, 8, and 32 pixels.

Figure 4.5 (second image from the left) illustrates what the recovered image would look like if there were no errors in the cache channel.

Varying Payload Size

First, we conduct an experiment to explore the trade-off between speed and accuracy when selecting the payload size. We measure the time required to recover the entire image after the attack setup has been completed (i.e., setup times are excluded). We recover images with packet sizes of 1, 4, 8, and 32 pixels. For each packet size, we perform 100 recovery attempts and report the median elapsed time and median error rate. Figure 4.5 (third to sixth images from the left) shows representative recovered images for each packet size.

The primary type of error introduced is a skewing of pixel locations in the recovered image. This skew occurs because the attack leaks a vertical column of pixels all at once – limited to the number of pixels that can fit within a single packet. If the attack misses a pixel in the signal then subsequent pixels become offset. Moreover, these errors become more common as the packet size increases.

Figure 4.6 illustrates quantitative results. We use the Levenshtein distance (Levenshtein, 1966) to measure the error rate and report the median value from 100 measurements. The median runtime to leak the entire 25px by 25px image across the three colour channels (red, green, and blue) is 116 s (16px per second) at the lowest speed to 7 s (267px per second) at the fastest. As expected, the median error rate increases as the speed increases, starting at 1% at the lowest speed and increasing to 10% at the highest speed.

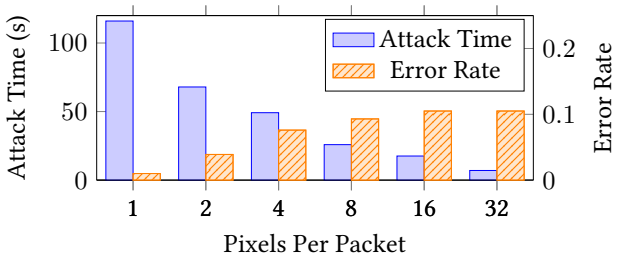


FIGURE 4.6: **Packet Size vs. Time & Error Rates:**
Data showing that increasing the number of pixels transmitted per packet reduces redundancy and therefore increases error rate, however it also significantly decreases attack time.

Identifying the Target Set

Thus far, our evaluation has assumed an attacker prepared to leak an arbitrary number of pixels. We now measure the time required to setup the attack.

Setup. To identify the target cache set, we use the technique described in Section 4.3.1. To validate our results, we patch Firefox to output the addresses of each eviction set and to output the address of the filter tables. We then use the methods of Maurice et al. (2015) to verify whether our attack correctly identifies filter memory accesses in the cache.

Method. Since false detections of the transmitted preamble are possible, we record multiple traces per cache set. If the preamble is detected at least once during these recordings, we mark the set as a candidate. We then record the same number of traces again from each candidate set. If the preamble is detected in at least three of these subsequent traces, we confirm the set as the target. This process is repeated 21 times for each number of recorded traces, and the median identification time and accuracy are reported in Table 4.2. The median time to identify the target set ranges from 372 to 1,257 seconds, with 48% to 77% of runs correctly identifying the target set on the first iteration through all eviction sets.

samples	Time (s)	Accuracy
40	1257.59±169	77%
30	999.56±152	67%
20	644.57±115	56%
10	372.39±60	48%

TABLE 4.2: **Time To Identify a Target Set:**
The median time in seconds to identify a target set and the portion of runs that identify the target set after searching through the cache once (Accuracy).

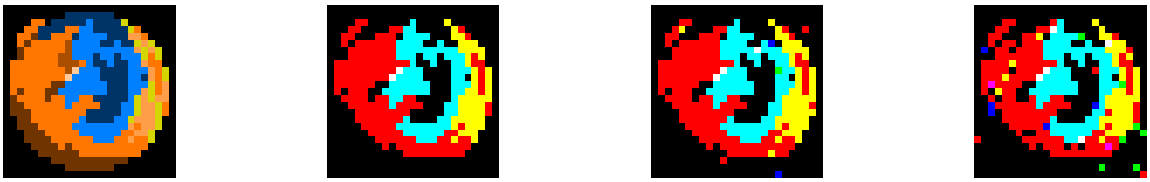


FIGURE 4.7: **Effect of system noise on leakage:**
From left to right: The original image, ideal leakage, a recovered image on a system without noise, and a recovered image on a system with noise.

System Noise

Finally, we evaluate our attack in a more realistic scenario where the victim has multiple browser tabs open. We use the same experimental setup as above but we open a new tab that plays a YouTube video. Figure 4.7 illustrates a representative qualitative result, showing a slight degradation in the quality of the recovered image. Over 100 runs, compared with earlier results, the median runtime increases to 476 seconds (4.09×) with a median error rate of 2.72% (2.68×).

Work	Side-Channel	Measurement Target	Mitigated	Speed
Stone (2013)	Filter Optimizations	Page Rendering	✓	10
Kotcher et al. (2013)	Filter Optimizations	Page Rendering	✓	<1
Andrysco et al. (2015)	Subnormal Floats	Page Rendering	✓	16
Kohlbrenner and Shacham (2017)	Subnormal Floats	Page Rendering	✓	60
Wang et al. (2023)	Power Consumption	Page Rendering	✗	3
Taneja et al. (2023)	Power Consumption	Page Rendering	✗	<1
This Work	Memory Accesses	Memory Accesses	✗	267

TABLE 4.3: **Comparing Pixel-Stealing Attacks:** We report the channel that information is leaked through, what each work measures to extract information from the channel, whether the attack has been mitigated, and the speed of the attack in bits per second (approximately).

4.3.3 Comparisons to Existing Works

Table 4.3 compares the speed of our attack with other cross-origin pixel-stealing attacks. All listed works leverage timing side-channel attacks on filters to extract cross-origin pixels. We compare the leakage channel used, the method each work employs to extract information from that channel, whether the attack has been mitigated, and the attack speed measured in pixels per second (using the figure reported by the work).

Earlier works exploit optimisations within the filters themselves, whereas later works, including our own, leverage leakage through microarchitectural side channels.

All previous attacks extract side-channel information by measuring the total time taken to render the page, thereby indirectly inferring the filter execution time. In contrast, our attack obtains side-channel information by measuring filter memory accesses. This fundamental difference enables us to achieve speeds that surpass the display refresh rate.

Like recent pixel-stealing attacks, our method remains unmitigated. Fortunately, generic countermeasures against cross-origin pixel-stealing attacks have been proposed, and these mitigation strategies would neutralize all such attacks, including the attack presented in this work. We discuss these countermeasures in greater detail in Section 4.6.

4.4 From Pixel Stealing to Text Stealing

We now demonstrate how to extend our basic pixel-stealing attack to create a text-stealing attack capable of recovering sensitive text content from third-party websites. We adapt previous work by Stone (2013), which targets monospaced fonts that are easily pixelated, to modern vector-based proportional-width fonts with kerning.

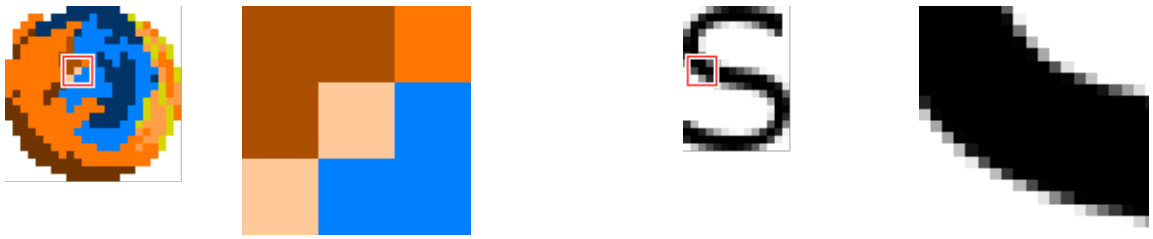


FIGURE 4.8: **Stretching Pixel vs. Vector Content:**
Pixel-based content remains pixelated after stretching (left), whereas vector-based content is resampled and remains smooth (right). Both regions in the original images are the same size: 3×3px.

Text Rasterization. Our pixel-stealing attack scales an image to isolate and amplify individual pixels. Pixel-based content can be scaled in a way that preserves the discrete structure of the original pixels (Figure 4.8 left). In contrast, vector-based content, such as text, is represented using mathematical expressions that are sampled to generate an image. When vector graphics are scaled, the scaling is applied to the underlying mathematical expression, and the image is then re-sampled at display resolution (Figure 4.8 right). Owing to this difference, applying transformations to isolate the colour of a single pixel is significantly less effective for text.

Pixel Stealing on Vector-Based Content. To illustrate the problem, we perform our pixel-stealing attack on a classic pangram and present the recovered text in Figure 4.9.

Compared with pixel-based content, the recovered vector-based text shows considerably more visual degradation, even when the attack recovers one pixel at a time. Despite the visual degradation, the phrase remains readable owing to the redundancy inherent

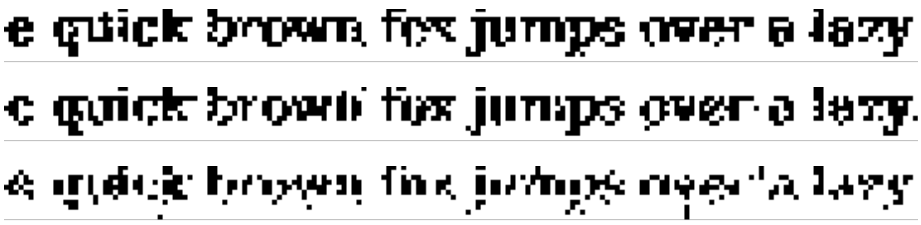


FIGURE 4.9: **Naive Text Stealing Results:**
Recovery of the phrase 'the quick brown fox jumps over a lazy dog' at 1, 2, and 4 pixels per packet.

in English text. However, this redundancy is often absent in the types of sensitive information attackers typically target, such as names, identification numbers, or passwords, which are short and contain little to no redundancy.

A natural solution to this problem is to sample the image at a higher resolution to minimise the degradation. However, doubling the resolution requires four times as many samples, because the number of pixels grows with the square of the side length. We ask: *Is there a better technique to extract text?*

Stone's Method. We answer this question in the affirmative by building upon the work of Stone (2013). Stone uses a pixel-stealing attack to recover text character by character. Rather than naively leaking the entire image of the character, Stone carefully chooses pixels to reveal so that each revealed pixel eliminates roughly half of the possible characters. Unfortunately, Stone's technique was designed to leak rasterised monospaced fonts, not proportional-width fonts that are more commonly used to display text.

Choosing a 'Pixel'. We begin by addressing the challenge of identifying the colour of 'pixels' in vectorised content. The key observation that enables the adaptation of Stone's method to vector-based text is that we can isolate arbitrary rectangular regions of the text, including rectangles with non-integer coordinates. If we ensure that these rectangular regions are entirely inside or entirely outside a character, then that rectangle will only contain a single colour and we will not get any visual artefacts after scaling. Figure 4.10 illustrates example regions for the letters 'A' through 'F'.

FIGURE 4.10: **Exmaple Regions:**
A region that lies entirely inside or outside each of the letters 'A' through 'F' when rendered using the MS Sans Serif font.

Finding Regions. We identify such regions in an offline preprocessing step using Selenium. [3] We randomly choose x and y coordinates and use a fixed width and height – on the scale of one-millionth of a pixel. We render each character and scale this region to the size of the screen. Finally, we take a screenshot of the browser then verify that the screenshot contains only a single colour and is artefact free.

After collecting enough regions to identify each character, we use a greedy search algorithm to minimise the number of regions. We select the region which provides the most information, the region that best splits the alphabet into two equal sets of characters, then store it in a list. We repeat this process until every character can be uniquely identified. While this approach does not guarantee the theoretical minimum number of regions, in practice, we find that it produces a small list of regions suitable for the attack.

Kerning. While this approach enables us to recover the first character, the location of the next character is hard to predict. Specifically, font rendering engines feature *kerning* and will move some characters closer together to achieve a better visual appearance. With kerning, the position of a character depends on the preceding characters. Although kerning can involve combinations of three or more characters, this is rare and so we restrict our analysis to the common case of two-character kerning.

To account for kerning during region collection, we randomly insert a character before every character in the alphabet. This change yields a list of regions for every possible pair of characters. We then modify our recovery algorithm to leak subsequent characters by

[3] *https://www.selenium.dev/*

using regions obtained with kerning – specifically, we use regions that begin with the previous character.

4.4.1 Text Stealing Results

We demonstrate our text-stealing technique in a PoC attack that recovers cross-origin text content – the victim's Wikipedia username. Since this is a cross-origin attack, we deploy our two-page architecture as described in Section 4.1 and manually interact with the page to allow the attack to open a second page in a new tab.

Identifying Username Location. To steal the username, we first need to locate it on the page. We start by manually opening the page in developer mode and using `getBoundingClientRect` to find the rough location of where the username begins. Then, we use Selenium to automatically refine the location by applying small adjustments and simulating the attack using screenshots. This is necessary because the resolution of `getBoundingClientRect` is not high enough to accurately locate the username.

Stealing the Username. Finally, we execute the attack to validate that we can successfully steal text in a cross-origin setting and recover the username of the account.

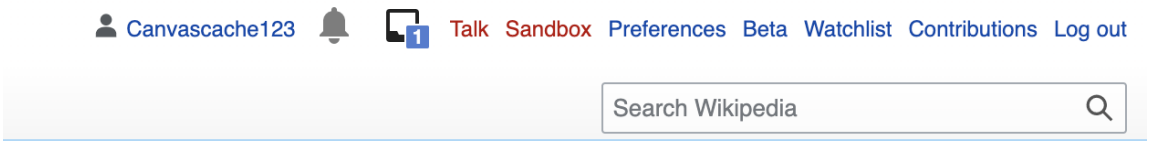


FIGURE 4.11: **Layout of Wikipedia Username.**

4.5 History Sniffing

In this section, we demonstrate how our pixel-stealing attack can be applied to history sniffing. We begin by describing a straightforward approach that tests individual URLs or small groups of URLs. We then introduce a batch testing method that improves the

sampling rate when the number of visited URLs is expected to be low relative to the total number of possible URLs.

```
.leak {
  display: inline-block;
  width: 1px;
  height: 1px;
  background-color: black;
  font-size: 0;
}

.leak:visited {
  background-color: white;
}
```

LISTING 4.4: **History Sniffing CSS Style:**
The style renders each URL as a single pixel. The pixel is black if the URL has been visited by the user before and white if it has been visited.

4.5.1 Straightforward History Sniffing

Previous history-sniffing attacks rely on the observation that the colour of a hyperlink can indicate whether it has been visited (Kotcher et al., 2013; Stone, 2013; Andrysco et al., 2015; Huang et al., 2020; O'Neal and Yilek, 2022). Our attack builds on the same principle.

To implement the history-sniffing attack, we first create a pixel whose colour reflects whether a specific URL has been visited. To achieve this effect, the attacker page includes a CSS style similar to the example shown in Listing 4.4.

The first rule styles each link as a 1×1 pixel box with a black background. The second rule uses the `:visited` selector to change the background to white, if the link points to a URL previously visited by the victim.

Once such a pixel is created to reflect whether a URL has been visited, we can apply our pixel stealing attack to recover the colour of the pixel and thereby determine whether the victim visited the website. We can leak multiple URLs simultaneously, by repeating these

steps to generate individual pixels with colours that reveal whether each corresponding URL has been visited. After revealing whether each URL has been visited, we update the `href` attribute of each link to point to a new URL, which prompts the browser to recalculate and update the appearance of each link.

4.5.2 Set Query Optimisation

History-sniffing attacks do not directly expose a victim's browsing history; instead, they leverage the browser as an oracle to determine whether a specific URL has been previously visited. To recover a large portion of the victim's browsing history, the attacker queries this oracle repeatedly. However, this process is computationally expensive, as the oracle must be queried for each URL independently.

To reduce this cost, we develop the `set query` method that exploits the underlying mechanisms of our pixel stealing attack to query over a thousand URLs at once. Recall that our pixel-stealing attack uses Prime+Probe to measure data-dependent memory accesses by a filter. However, because memory access occur much faster than we can perform Prime+Probe measurements, we had to reduce the speed of the filter by stretching the image.

Blurring Measurements. If we did not stretch the image, then the filter would process different pixels within a single Prime+Probe measurement 'blurring' the effects of several distinct memory accesses together. In each measurement, the attack detects access to a specific memory location but it cannot determine the number of accesses to that location or the order of accesses to that location.

Typically, blurring memory accesses together in this way is undesirable because it loses information. For example, a set of pixels that contains exactly one white pixel results in the same measurement as a set of pixels that contains several white pixels. In both cases, the attack detects that the location in memory for white pixels was accessed, but cannot determine the number of memory accesses or the order of memory accesses to that location. Because the attacker cannot distinguish between these two cases, they cannot determine which pixels are white, they only learn that one or more pixels are white.

Leveraging Blurring. While Prime+Probe cannot determine the number of memory accesses to a location, it can distinguish zero accesses from one or more accesses. The key insight is that if we expect most URLs to be unvisited, we can group these URLs together

and use a single measurement to confirm whether all the URLs are unvisited. Figure 4.12 illustrates example measurements with no visited links and some visited links.

Specifically, we encode unvisited links as black pixels and visited links as white pixels then monitor filter memory accesses. If no white pixels are observed, then we conclude that none of the links have been visited and move onto the next set to query. Otherwise, we split the set into two subsets and perform the same measurement on each subset. If a subset contains at least one visited URL, we continue to recursively apply this process until we either reach a subset without any visited URLs or a subset that contains only a single URL that is visited.

When exactly one URL is visited, this binary search allows us to identify it with $\mathcal{O}(\log n)$ samples. However, as the number of visited URLs increases, the required samples approaches $\mathcal{O}(n \log n)$, making it efficient in such cases to use the *simple* method, which queries each URL individually and always identifies the visited URLs in $\mathcal{O}(n)$ samples.

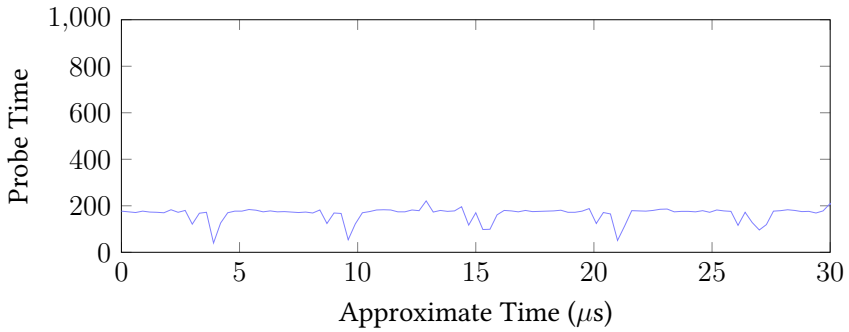
Method Selection. We address this issue by dynamically switching between the *simple* and *set query* methods based on whether the website is in the Alexa Top 1,000 most visited websites. For websites in this list, we assume a high likelihood that the URL has been visited and use the simple method. For all other websites, we use the set query method.

4.5.3 Experiment Description

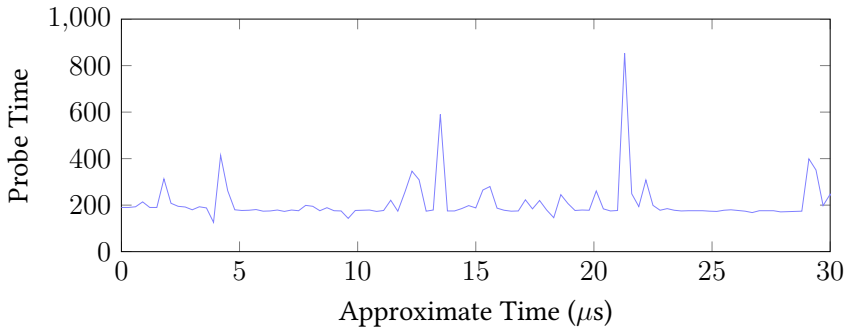
In this section, we describe the experiments conducted to evaluate the performance of our history-sniffing attack.

Fabricating History. We begin by fabricating browsing history. We sample from the Alexa Top 50,000 websites, so that popular websites are more likely to be included in the fabricated history. [4] We then use Selenium to populate the browser history by opening each website. Since not all websites will be stored successfully in the browser history, either because the website failed to load or because it redirected the browser elsewhere, we verify the list of websites inserted into the history by simulating the attack. Specifically, we navigate to a webpage that renders the list of websites as black or white pixels using the

[4] We include websites in the history with a probability of roughly $\frac{1}{n}$ where n is the position of the website in the Top 50,000.



(A) No Visited Links



(B) Some Visited Links

FIGURE 4.12: **History Sniffing Memory Access Patterns:** Access Patterns for the Set-Query Attack.

style shown in Listing 4.4 then take a screenshot to obtain the ground truth for evaluating the attack.

History Sniffing. The history-sniffing attack is performed over the same set of 50,000 websites. As described earlier, we use the *simple* method for the top 1,000 websites and split the remaining 49,000 websites into batches of 1,000 and use the *set query* method to recover them.

4.5.4 Results

Table 4.4 summarises the results of our experiments. For each test, we vary the number of samples and measure its affect on runtime, recall (the proportion of websites in the victim's history that were identified by the attack), and accuracy (the proportion of websites identified as visited that were in the victim's history). We only evaluate the *set query* method, as the *simple* method's performance is identical to regular pixel stealing analysed in Section 4.3.2.

4.6 Countermeasures

Total Cookie Protection. Total Cookie Protection is a new feature that enforces stricter control over HTTP cookies by ensuring that each requested cookie is only sent to the domain that originally set it. Requests that violate this policy are blocked. Mozilla enabled Total Cookie Protection globally for all users on June 14, 2022. This update applies to Firefox versions starting from Firefox 101 and all versions of Firefox Nightly but does not affect any versions of Firefox ESR. While this feature effectively mitigates all cross-origin pixel-stealing attacks, it does not impact the history-sniffing attack, because the rendering of visited links is independent of cookies. Safari implements a similar countermeasure called Intelligent Tracking Protection, which also prevents cross-origin pixel-stealing attacks.

Constant Time Programming. Constant-time programming is a coding style designed to prevent the use of constructs that unintentionally leak secret information (Osvik et al., 2006; Brickell et al., 2006; Barthe et al., 2014). Specifically, secret values must not be used in the conditions of control flow statements, as memory access addresses, or as arguments to variable-time instructions (e.g., division). While these constraints are highly effective at preventing side-channel leakage, they often introduce significant performance overhead and can be difficult or impossible to implement correctly in high-level languages. We recommend this approach to browser vendors for any fallback filters that are executed on the CPU.

frame-ancestors. The `frame-ancestors` directive of the `Content-Security-Policy` is a widely supported feature that allows developers to prevent their webpages from being

loaded within `iframe` s, thereby blocking malicious embedding. While this directive does not prevent information leakage from the webpage itself, it stops malicious sites from embedding and leaking content from otherwise safe victim webpages. Considering that this technique also protects against other attacks such as click-jacking (Rydstedt et al., 2010; Huang et al., 2012), we recommend that all website operators implement it. For website operators that need to support embedding within arbitrary webpages, removing sensitive information when the content is embedded is advisable.

4.7 Limitations & Future Work

Pixel Stealing. Our pixel-stealing attack is limited to environments where SVG filters are executed on the CPU rather than the GPU. In Firefox, this condition applies to all environments. In Chrome, the attack is only applicable if the system is listed on the Software Rendering List (Chromium Project, 2023). Additionally, our attack is limited to Intel CPUs featuring inclusive caches. In principle, the attack could likely be extended to CPUs with non-inclusive caches, potentially by exploiting coherence directories (Yan et al., 2019), we leave this extension for future work.

Text Stealing. Our text-stealing attack is limited to scenarios where the attacker can correctly guess the font used by the user. If the user changes the font or uses an unknown font, they are not vulnerable. While the technique can theoretically extend to languages with very large alphabets, such as Mandarin, in practice, the attack becomes less practical as alphabet size increases.

Cross-Origin Content. Our attack assumes that cross-origin content does not employ security measures such as `COEP`/`COOP`, `X-Frame-Options`, or the `frame-ancestors` directive. Unfortunately, many websites still do not implement these protections and remain vulnerable to our attack (Foundation; The HTTP Archive; Helme; Lavrenovs and Melon, 2018; Karopoulos et al., 2021; Buchanan et al., 2018). In addition, we assume that the browser either does not support or has not enabled Total Cookie Protection or similar countermeasures.

Theoretical Maximum Bitrate. Our pixel-stealing attack, in its maximum pixels-per-packet configuration, has a theoretical maximum throughput of 1920 bits per second – 32

bits per packet and 60 packets per second. In practice, the attack achieves a maximum speed of 267 bits per second. While this practical throughput makes it the fastest side-channel-based pixel-stealing attack to date, it still reaches only about 14% of the theoretical maximum.

This limitation arises partly because the attack cannot dedicate all its time to recording cache activity; it must also allocate time to analyse recorded traces and to identify and extract packets. Our measurements indicate that approximately half of the attack runtime is spent analysing cache traces.

In principle, the analysis could be offloaded to a separate thread to allow more continuous recording of cache activity. However, naive attempts to implement this simply shifted the bottleneck to message-passing overhead between threads. We leave a more thorough investigation of efficient multi-threaded collection and analysis methods for future work.

In addition to analysis overhead, we find that the attack fails to receive a quarter of all packets on average, even after accounting for missed packets transmitted during analysis. We attribute this loss to system noise obscuring the preamble, making it unrecognisable by our signal processing pipeline. Although deploying a more robust signal processing pipeline with advanced techniques could capture more packets, it is unclear whether the benefits would outweigh the additional packet loss due to increased analysis time. We leave a more thorough investigation of this trade-off to future work.

4.8 Conclusions

In this chapter, we present Pixel Thief, a cache-based pixel-stealing attack targeting Firefox's SVG filtering engine. Despite several mitigation efforts, we demonstrate that pixel stealing remains not only possible but also highly practical.

We develop an asynchronous architecture that allows the attacker to measure cache leakage in parallel with SVG filter execution. This approach enables multiple measurements per filter execution, increasing the data rate and ultimately overcoming the limitations imposed by screen refresh rates. In addition, we distribute the asynchronous architecture across two webpages to bypass cross-origin isolation. This technique may be applicable to other microarchitectural attacks, casting doubt on the effectiveness of cross-origin isolation as a countermeasure for microarchitectural side-channel attacks.

Fortunately, browser vendors have already begun isolating cookies from `iframes`. Although these features are mainly designed to mitigate tracking threats, they also help prevent cross-origin pixel-stealing attacks. Unfortunately, same-origin pixel-stealing attacks, such as the history-sniffing attack presented in Section 4.5, remain unaffected; however, cryptographic constant-time techniques can likely be applied to SVG filters with minimal performance impact to address these attacks.

Thus far, this thesis has focused on traditional microarchitectural side-channel attacks, where the target program, the browser in this case, unintentionally leaks sensitive data because the program exhibits data-dependent behaviour which the adversary indirectly measures. The next two chapters investigate transient-execution attacks and their ability to leak sensitive data even when the program does not directly exhibit data-dependent behaviour.

Fixed Sample Count

Samples	Runtime (s)	Recall	Precision
1	26.27	10%	3%
2	22.75	0%	0%
3	91.21	32%	26%
4	81.62	36%	9%
5	158.26	42%	42%
6	165.71	47%	26%
7	286.61	40%	60%
8	254.34	55%	47%
9	354.77	52%	59%
10	354.62	52%	56%
11	441.97	47%	57%
12	420.86	64%	65%
13	525.45	47%	80%
14	477.48	68%	73%

TABLE 4.4: **Set Query Accuracy Results:**
Accuracy of the set query method using different measurements counts and a majority vote. Recall is the proportion of websites that the attack was able to recover from the victim history. Precision is the proportion of websites claimed by the attack to be in the victim history that were actually in the victim history. Reported time only includes the time to recover the complete history and excludes pre-attack setup time.

Chapter 5

Mounting a Transient Execution Attack on Modern Browsers

To improve performance, modern processors include branch prediction units that predict the outcome of branch instructions. When a branch instruction is encountered, the branch predictor predicts whether the branch will be taken or not, allowing the processor to continue executing the program speculatively. If the prediction is correct, the processor gains a performance advantage by avoiding the need to wait for the branch outcome. However, if the prediction is incorrect, the processor will need to squash the incorrect execution and re-execute the program along the correct path.

Kocher et al. (2019) were the first to demonstrate how to exploit this behaviour. Specifically, they showed how an adversary can use JavaScript to bypass bounds checks on array accesses, resulting in out-of-bounds memory access. This ability to bypass bounds checks via JavaScript posed a significant problem for many web browsers, as they commonly relied on these bounds checks and other software-level security features to isolate websites rendered within the same process. Recognising the risk posed by transient-execution attacks, browser vendors have largely adopted new security architectures that isolate websites into separate processes (Reis et al., 2019), even when these sites are embedded together on the same webpage via `iframes`. While these process boundaries do not prevent malicious JavaScript from performing transient-execution attacks, they limit accessible data to within the process.

Chrome was the first browser to implement such an architecture owing to its existing multi-process architecture. To further limit accessible memory with out-of-bounds memory accesses, Chrome partitions the address space to isolate each webpage.

These security controls use the same-site policy to determine when to isolate webpages from each other. Unlike the same-origin policy which considers the entire domain, the same-site policy only considers the final portion of the domain – the Effective Top Level Domain Plus One (eTLD+1). Previous studies have demonstrated vulnerabilities that lie in the gap between these two policies (Squarcina et al., 2021; Bortz et al., 2011) which naturally leads to the question: *Can transient-execution attacks exploit the same gap on modern browsers despite deployed countermeasures?*

This chapter addresses this question by presenting Spook.js, a transient-execution attack capable of extracting sensitive information from co-resident webpages despite site isolation and address-space partitioning. Section 5.1 outlines the challenges involved in mounting transient-execution attacks on modern browsers and explains how we overcome them. Sections 5.2 and 5.3 present several attack scenarios in which usernames, passwords, credit cards numbers, and other personal information are extracted from various websites, password managers, and extensions. Section 5.4 shows that other Chromium-based browsers are vulnerable to Spook.js with minimal adaptation. Finally, Sections 5.5 and 5.6 conclude the chapter with a discussion of countermeasures and limitations.

5.1 Spook.js: Mounting Transient Execution Attacks in Chrome

We now present Spook.js, a JavaScript-based transient-execution attack capable of recovering information across security domains running concurrently in the Chrome browser. In addition to bypassing all side-channel countermeasures deployed in Chrome (such as low-resolution timers), Spook.js overcomes several key challenges that remained unresolved in previous works.

Site Isolation. The core principle behind site isolation is that it separates the attacker and victim pages into separate processes. However, because this separation is enforced at the site level (as opposed to the origin level), there is a gap between what the website developers consider to be a website and what the countermeasures consider to be a website. We introduce several methods that exploit this gap to consolidate different websites into the same process.

Address Space Isolation. Chrome uses pointer compression to limit the size of pointers which also limits the amount of memory accessible to each pointer – even in cases where an attacker can control the pointer. Chrome leverages this limitation in combination with carefully arranging the address space to limit out-of-bounds memory accesses. We intentionally misuse objects of one type with operations specialised for another type to trigger transient type confusion to bypass this countermeasure.

Deoptimisation. As we use objects of the wrong type, the JavaScript engine will 'de-optimise' the specialised code by replacing it with non-exploitable generic code capable of handling both types. We adapt speculative hiding techniques to the browser to avoid this deoptimisation.

Speculation Depth. Overcoming address space isolation and avoiding deoptimisation requires executing many instructions transiently. We use evictions from the LLC to induce a delay long enough to execute these instructions.

5.1.1 Website Consolidation

To mount transient execution attacks, both the attacker and target websites must reside within the same address space. Chrome's site isolation aims to prevent cross-site attacks by segregating websites into separate processes. However, Chrome allows websites to be consolidated into the same process according to the same-site policy. Specifically, if the two websites share the same eTLD+1 domain then they are eligible for consolidation. We now discuss methods to force consolidation between attacker and victim pages.

iframes. We first observe that consolidation can be achieved by embedding iframes within a page from the same site. While this method is effective, it cannot be used on pages that prevent rendering inside iframes, for example, by setting the `X-Frame-Options` header to deny. Because many attacks exploit iframe vulnerabilities, setting this option is a widely recommended security practice and is employed by many websites.

Memory Pressure. When under memory pressure, Chrome reduces memory usage by consolidating pages from different tabs, provided the pages originate from the same site. Furthermore, once consolidation occurs, Chrome adds newly opened pages from into existing processes rather than creating new processes for those pages, again provided the pages originate from the same site. However, it is difficult to build an attack around this

consolidation technique because the attack does not have enough control over memory consumption across the entire browser.

Opening Windows. Finally, we observe that Chrome consolidates pages opened using the `window.open` API when these pages share the same site. Although lacking stealth, this method is reliable and does not suffer from the same limitations as the previous consolidation techniques.

Experimental Results. In all cases, these methods allow an attacker operating on one subdomain (e.g., *attack.example.com*) to consolidate their pages with content from a separate subdomain (e.g., *sensitive.example.com*). We evaluate the effectiveness of each method using Chrome 89.0.4389 on a machine with an Intel i7-7600U CPU and 8 GB of RAM, running Ubuntu 20.04. We find that both the iframe and opening window methods will always consolidate pages from the same site.

To evaluate the memory pressure method, we open several websites in different tabs. We find that opening 17 websites causes Chrome to begin consolidation. However, the threshold for consolidation depends on the machine's memory size. On a machine with more memory we need to open more websites (e.g., 33 websites on a machine with 16 GB of RAM).

5.1.2 Breaking Address Space Isolation

Chrome employs a pointer compression technique that represents pointers as 32-bit offsets from a fixed base address (Sheludko and Solanes, 2020). This base address defines an *isolate* within which memory is allocated. Because pointers are represented as 32-bit offsets, instead of a full 64-bit address, the amount of memory accessible to each pointer is limited. Chrome leverages this limitation as a security feature by carefully arranging the address space so that memory accesses through compressed pointers are constrained to their respective isolate.

In this section, we overcome this countermeasure using transient type confusion. Specifically, we trick Chrome into accessing a malicious object as a typed array. Because typed arrays use 64-bit pointers to refer to their underlying buffers, we can use the malicious object to access arbitrary 64-bit addresses, bypassing Chrome's partitioning countermeasures. While type confusion attacks have been previously demonstrated in the literature (Kiriansky and Waldspurger, 2018; Hadad and Afek, 2018; Kirzner and Morrison,

2021), to the best of our knowledge, this is the first demonstration of transient type confusion attacks against the modern Chrome browser.

```
UInt8Array-access(array, index){
  // check the type of array
  if(array.type !== UInt8Array){
    goto interpreter // handle wrong type
  }

  // compute array length
  len = array.length

  // check length is in bounds
  if (index >= len) {
            goto interpreter // Handle out of bounds
  }

  // compute pointer to backing storage
  ptr = array.external + ((array.base + heap_ptr) & 0xFFFFFFFF)

  // do memory access
  return ptr[index]
}
```

LISTING 5.1: **Pseudocode for Array Accesses:**
Pseudocode of operations performed by Chrome's JavaScript engine during array accesses.

Array Indexing. Consider the following JavaScript: `array[index]`. There are several valid meanings for this expression that depend on the types of `array` and `index`. Chrome optimises this expression by speculating that these types never change, enabling chrome to replace generic code capable of handling any possible combination of types with specialised code optimised to handle only a specific set of types.

For example, if `array` is a `Uint8Array` and `index` is an integer, then Chrome replaces the generic code with specialised code similar to Listing 5.1. The code first checks whether `array` is a `Uint8Array` (Line 3) then checks whether `index` is inside of the array (Line 11). If both checks are successful, the code constructs a pointer to the array buffer (Line 16) and dereferences it with `index` (Line 19). If either check (Lines 3 or 11) fails, control flow returns to the interpreter, which handles the more complex generic cases.

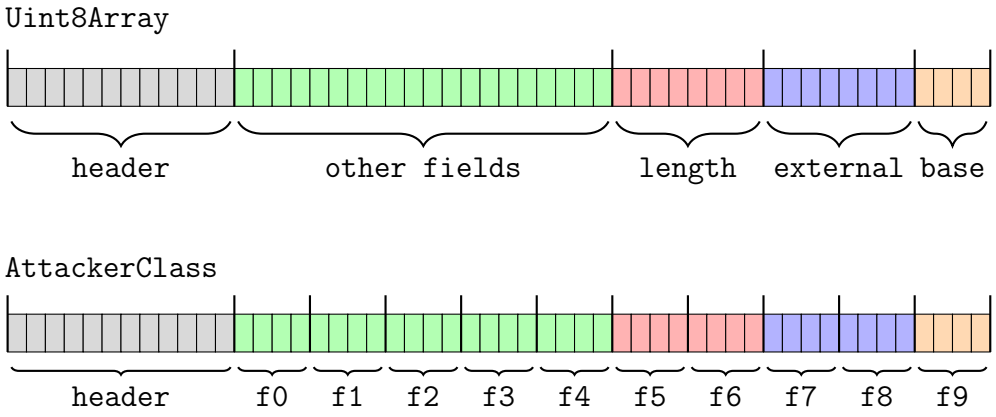


FIGURE 5.1: **Compariing Memory Layout:**
The memory layout of `Uint8Array` (top) and `AttackerClass` (bottom). Ticks denote the boundaries of fields, rectangles within each field denote a byte, and colours are used to highlight field alignment across types.

Array Layout. Figure 5.1 (top) illustrates the memory layout of a `Uint8Array` object. The layout begins with a header that identifies the object type and includes other metadata. The header is followed by several fields that define the array properties. Of particular interest are the `length`, which tracks the number of elements in the array, and the `external` and `base` fields, which together form a pointer to the underlying array buffer. [1]

A Malicious Memory Layout. Recall that the goal of our attack is to trick Chrome into accessing a malicious object as a typed array. Figure 5.1 (bottom) illustrates the layout of this malicious object. `AttackerClass` consists of ten fields named `f0`–`f9`. Because

[1] Two fields are used for legacy reasons.

these fields are JavaScript properties, Chrome uses 32-bits to represent each field. Notably, when we compare the layouts of `AttackerClass` with `Uint8Array` the `f5` and `f6` fields align with `length`, `f7` and `f8` with `external`, and `f9` with `base`.

Type Confusion. Our attack works by training the processor to predict the outcome of type checking branches within an array access (Listing 5.1 Line 3). When we later invoke an array access on a malicious object, the processor follows this prediction which causes the JavaScript engine to access the malicious object as though it were a `Uint8Array`, treating the values of `f7`–`f9` as though they defined a pointer to the array's underlying buffer. Because we control every field of the object, we can manipulate the address to access arbitrary process memory. Appendix A describes the precise method we use to craft arbitrary pointers.

5.1.3 Avoiding Deoptimisation

The previous section described how we cause the processor to mispredict a type checking branch within an array accesses to trigger an out-of-bounds memory access. Eventually the processor will detect this mispredicted branch, squash the resulting incorrect execution, and restart execution with the correct branch outcome. In this case, the branch correctly identifies that our object is not a `Uint8Array` and directs control flow into the JavaScript engine, which 'deoptimises' the array access by replacing it with a non-exploitable generic code path.

We avoid this deoptimisation by adapting speculative hiding techniques (Lipp et al., 2018; Mambretti et al., 2020; Göktas et al., 2020; Canella et al., 2019) to the browser context. Specifically, we insert an additional branch that has the same behaviour as the type checking branch. In the training phase, both branches are taken, and in the attack phase, both branches are predicted as taken. However, when the processor detects the misprediction, it will restart execution at this additional branch instead which enables us to direct control flow back into the attack.

Listing 5.2 presents the JavaScript code for the attack, which consists of four main components, described below.

```
// Setup
let objects = new Array(128)

for (let i = 0; i < 64; i++) {
  objects[i]      = new Uint8Array(0x20)
  objects[64 + i] = new AttackerClass(i)
  collectGarbage()
}

let [index, set] = findSuitableObject(objects)
objects[index].f0 = 1

// Training
for(let i=0; i<10000; i++) gadget(0)

// Attack
objects[index].f5 = 1 // length (bottom 32-bits)
objects[index].f6 = 0 // length (top 32-bits)
objects[index].f7 = Lower32BitsOfAddress
objects[index].f8 = Upper32BitsOfAddress
objects[index].f9 = 0
set.evict() // Evicts object[index]
gadget(index)
side_channel.receive()

// Gadget
function gadget(i){
  if(i < objects[index].f0) {
    let object = objects[i]
    let value = object[0]
    side_channel.send(value)
  }
}
```

LISTING 5.2: **Pseudocode for Speculative Type Confusion:**

Setup. Lines 4–8 initialise an array of objects assigning some entries to `Uint8Array` instances and others to `AttackerClass` instances. Line 7 triggers Chrome's garbage collector, by allocating many large buffers and allowing them to go out of scope, which compacts the heap and reallocates previously initialised objects into contiguous memory locations. Line 10 identifies a suitable malicious object. Finally, Line 11 sets a value that we will use to avoid deoptimisation later.

Training. The training stage has two goals. First, it provides ample opportunity for Chrome to specialise the array access code in `gadget` to the code in Listing 5.1. Second, it trains the processor to predict specific outcomes for branches within `gadget` (including the branches in the array access). Line 14 achieves these goals by calling `gadget` 10,000 times with an index to a `Uint8Array`.

Attack. Lines 17–21 configure fields `f5` through `f9` which correspond with the `length`, `external`, and `base` fields of a `Uint8Array`. Specifically, we set `length` to 1, `external` to the target 64-bit address, and `base` to 0. Line 22 evicts the header of the malicious object from the cache, then Line 23 calls `gadget` with the index of the malicious object. After the gadget returns, Line 24 retrieves the contents of memory at the target address (leaked on Line 31).

Gadget. The core of the attack occurs within `gadget`. Line 28 is the additional branch we use to avoid deoptimisation. Line 30 is the vulnerable array access, specialised for `Uint8Arrays` because of the training phase. During the attack phase, `gadget` triggers type confusion by performing the vulnerable array access on an instance of `AttackerClass`. The contents of the selected address are loaded into `value` which is leaked on Line 31 through a side channel. After some time, the processor detects the misprediction inside of the array access and continues execution at Line 28 which immediately exits `gadget`.

5.1.4 Obtaining Deep Speculation

The previous sections describe how to bypass address space isolation and avoid deoptimisations, however these techniques require many instructions to be executed transiently. To provide enough time for the processor to execute instructions beyond the type checking branch, we delay the evaluation of the branch by evicting the header (which contains the type identifier) of our object from the LLC. However, we need to still use the object as

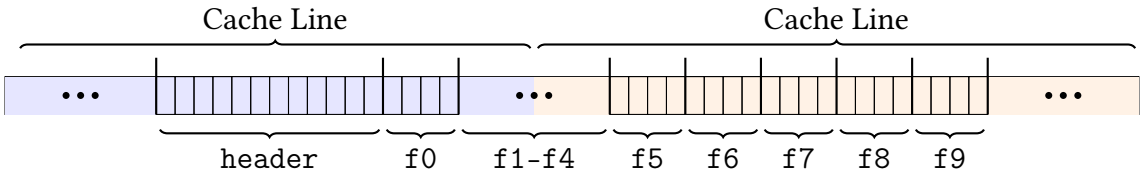


FIGURE 5.2: **Cacheline Overview:**
A malicious object split across cache lines. The location of the split must occur between the `f0` and `f5` fields. Ticks denote the boundaries of fields, rectangles within each field denote a byte, and colours are used to highlight which cache line data resides in.

a `Uint8Array` during type confusion. Specifically, we need to evict our object's header from the cache but retain the fields `f5`–`f9` in the cache. We do this by placing our object over two cache lines with the boundary between these cache lines somewhere between the `f0` and `f5` fields (Figure 5.2).

Use of Memory Compaction. In a previous section, we described how the attack triggers Chrome's garbage collection after allocating instances of the `Uint8Array` and `AttackerClass` types. As a side effect, garbage collection compacts the heap by reallocating these objects into a contiguous memory layout. Because the sizes of an `AttackerClass` object and a cache line are not multiples of each other, compacting the objects together changes where objects begin within the cache lines, guaranteeing the existence of at least one object with `f0` and `f5` fields that span two cache lines.

Finding Suitable Objects. Listing 5.3 presents the code used to identify a suitable instance of `AttackerClass`. Line 1 generates eviction sets for the entire LLC using the method from Vila et al. (2020). Line 4 iterates over each instance of `AttackerClass` and Line 7 iterates over each eviction set. Lines 8–10 and Lines 12–14 implement the logic for detecting whether the eviction set can successfully evict the `f0` field while leaving the `f5` field unaffected. Once a suitable object is found, the code returns the index of our object and the eviction set to evict it.

Because we cannot directly access the header of any object, we use the first field (`f0`) instead. Since the object is smaller than a cache line, if `f0` and `f5` are on separate cache lines then `f0` must be on the same cache line as the header.

```
EvictionSets = GenerateEvictionSets()

function findSuitableObject(objects) {
for (let index = 64; index < objects.length; index++) {
  let candidate = objects[index]

  for (set in EvictionSets) {
    access(candidate.f0)
    set.evict()
    let x = isCached(candidate.f0)

    access(candidate.f5)
    set.evict()
    let y = isCached(candidate.f5)

    if (x && !y) {
      return [index, set]
    }
  }
}}
```

LISTING 5.3: **Pseudocode for Finding Objects that Straddle Cachelines.**

Precision Timing. To determine whether a field has been evicted from the cache, we measure the time taken to access it. However, in an effort to reduce the threat of microarchitectural attacks, Chrome has reduced the resolution of its timer API (Chromium Project, 2016b). We follow the approach of Schwarz et al. (2017), and implement a counting thread using web workers and `SharedArrayBuffer`.

L1 vs LLC Evictions. Google's leaky.page transient-execution PoC (Google, 2021) uses L1 evictions to evict fields from the cache. While constructing L1 eviction sets is far simpler and is more reliable than constructing LLC eviction sets, our attack requires

a greater delay after misprediction. If we evict our object's header from the L1 cache, without evicting it from the LLC, our attack consistently fails.

We believe this failure is due to the size of the speculation window – the number of instructions that can be executed between misprediction and the processor squashing the resulting incorrect execution. When `f0` is evicted from the L1 cache, it will be served from the L2 cache within approximately 10 cycles. However, if `f0` is evicted from the LLC, it will be retrieved from main memory in over 100 cycles. We believe that our attack requires this longer retrieval time so that it can transiently execute more instructions, requiring the need for complex LLC eviction techniques. We leave further investigation of this effect to future work.

5.1.5 End-to-End Attack Performance

The previous sections have described a combination of techniques that enable an attacker to recover the contents of arbitrary memory addresses within the process. We now evaluate the effectiveness of these techniques across several generations of processors, including CPUs made by Intel, AMD, and Apple.

Attack Setup. On Intel and Apple processors, we run Spook.js on unmodified Chrome version 89.0.4389.114. For benchmarking, we initialise a 10 KB memory region with known random content and then use Spook.js to leak it.

Intel and Apple. Table 5.1 summarises our findings, averaging over 20 attack attempts. Spook.js leaks data at a rate of 500 b/s on Intel processors ranging from the 6th to the 9th generation, with an accuracy exceeding 96%. On the Apple M1, Spook.js achieves a leakage rate of 450 b/s with 99% accuracy.

AMD Zen. Unfortunately, we are unable to construct LLC eviction sets on AMD's Zen architecture. As Spook.js relies on the larger speculation window provided by LLC eviction, we cannot run end-to-end Spook.js experiments on AMD systems. To evaluate the core speculative type confusion attack without eviction sets, we modify V8 to expose the `clflush` instruction. With this configuration, Spook.js achieves a rate of around 500 b/s, indicating that if an efficient LLC eviction method is developed, Spook.js could be applied to AMD processors too. We leave the creation of such eviction techniques to future work.

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

TABLE 5.1: **Spook.js Performance on Various Architectures:** Performance of Spook.js on various processors. Speed is in bits per second. Error is the percentage of bits that were incorrectly recovered.

5.2 Attack Scenarios

In this section, we explore the real-world implications of Spook.js by investigating several scenarios where the attack can extract secret or sensitive information.

Experimental Setup. All experiments in this section are conducted on a ThinkPad X1 laptop with an Intel i7-7600U CPU running Ubuntu 18.04. Unless otherwise noted, we use an unmodified Chrome version 89.0.4389. All Chrome settings remain at their default configurations, including all built-in countermeasures against side-channel attacks.

5.2.1 Website Identification

In our first scenario, we assume the attacker hosts a malicious page containing Spook.js code on a public hosting service. The attacker then convinces the victim to open an unrelated page from the same hosting service, for example, the victim's personal page. While most pages on the hosting service are publicly accessible, the information about which pages the victim currently has open is private and should remain inaccessible to the attacker.

Attack Setup. To demonstrate how Spook.js compromises the victim's privacy, we conduct the attack on *bitbucket.io*, a Git-based hosting service. We host a malicious webpage containing the Spook.js code on *bitbucket.io* and create three sample personal pages, as shown in Figure 5.3 (top). Following Bitbucket's naming convention, the URLs of all four pages follow the pattern *username.bitbucket.io* , making them eligible for consolidation. The ground truth usernames for our three sample personal pages are: spectrevictim, lessknownattacker, and knownattacker.

Experimental Results. After opening the four websites in separate tabs, we consolidate all Bitbucket pages into a single renderer process then use Spook.js to leak the memory space of that renderer process. By analysing the results, we successfully recover a list of URLs corresponding to the tabs currently being rendered (Figure 5.4). Although the content of our sample victim pages is publicly accessible, the list of bitbucket.io websites simultaneously viewed by the user is private and should not be exposed to a malicious page hosted on *bitbucket.io*.

5.2.2 Recovering Sensitive Information

In the second scenario, we examine a protected subdomain that displays private data to authenticated users. As an example, we leverage the structure of our university's website, which, at the time of writing, hosts its main page, single sign-on (SSO) page, and internal portal pages under the same site as personal webpages.

Attack Setup. In collaboration with our University's IT department, we host the Spook.js code on a personal webpage. While logged in with the author's account, we open three pages from the human resources subdomain in separate tabs. Each page displays the author's contact details and direct deposit information (Figures 5.5 and 5.6 top). To protect the author's privacy, we edit the local copy of the DOM before executing the attack.

Experimental Results. After opening the three tabs, we also open the page hosting the Spook.js attack. Chrome consolidates all four pages into the same renderer process, allowing the attack page to recover sensitive data displayed on the other three pages (Figures 5.5 and 5.6 bottom).

5.2.3 Attacking Credential Managers

The previous chapter demonstrated how to use Spook.js to steal sensitive information embedded within pages hosted on the same site as an attacker controlled page. This section demonstrates the security implications of Spook.js on popular credential managers, which automatically fill in login credentials associated with a website, often without any user interaction. Furthermore, we show that credentials can be recovered even if the user does not submit them by pressing the login (or any other) button, as simply populating the credential fields places this sensitive data into the renderer process's address space.

Attacking Chrome. We use the experimental setup described for the university experiments, where both the login page and the internal attacker page are hosted by the university and rendered within the same process. We assume that the credentials are already saved in Chrome's password manager. Figure 5.7 illustrates the results of the experiment in which Chrome's password manager auto-fills the credentials (top), and we successfully recover these credentials without requiring any user interaction (bottom).

Attacking LastPass. We achieve similar results when using the same experimental setup but with LastPass version 4.69.0 to autofill passwords instead of Chrome's built-in manager. In addition to leaking credentials, we also extract other account usernames associated with the website. Figure 5.8 illustrates the results of this experiment in which LastPass enables the user to select which account to login as (top), and we successfully recover the list of account names (bottom).

One-Click Credential Recovery. We take the previous attack a step further by showing that credentials can sometimes be recovered as soon as the victim opens our malicious webpage without the need for additional tabs. This capability is enabled by two key observations: First, that the login page can be embedded within an iframe. Second, that LastPass fills user credentials automatically, without any visibility or interaction with the user. Figure 5.9 illustrates the results of the experiment in which we embed the login page within an invisible iframe (top) and successfully extract the credentials without the need for the victim to voluntarily or knowingly visit the login page (bottom).

Extracting Credit Cards. In addition to passwords, credential managers also manage credit card information, automatically filling the information when authorised by the user. Figure 5.10 illustrates the results of an experiment in which we successfully recover the

victim's card details (top) after they are populated on a payment page (bottom). We achieve similar results with both LastPass and Chrome's built-in credit card autofill features.

5.2.4 Attacking Tumblr

The previous section demonstrated how to use Spook.js to steal credentials that were automatically filled in by a credential manager. This section demonstrates the same technique on Tumblr, a popular micro-blogging platform with 327 million unique visitors as of January 2021 (statista, 2021).

Attack Setup. The Tumblr platform hosts user blogs under the domain *username.tumblr.com*, while the login page and account settings are hosted under *tumblr.com*. This architecture relies on enforcement of the same-origin policy to prevent cross-site attacks. Unfortunately, because these pages all share the same site, Chrome will allow them to be consolidated into the same process.

Although users cannot freely add JavaScript to blog posts, or otherwise inject it into the *tumblr.com* domain, they can customise their own blog's theme using HTML. More specifically, Tumblr's Cross-Origin Resource Sharing (CORS) policy prevents importing scripts from different origins, and its Content Security Policy (CSP) disallows creating Blob objects. Despite these restrictions, Tumblr's CSP permits the use of data URLs and the eval function, enabling us to embed the attack code as inline JavaScript within a URL. For this attack we use Chrome version 90.0.4430.

Attack Results. Using a similar setup to the previous section, we create a malicious blog on Tumblr's platform containing the Spook.js attack code. We achieve consolidation in two ways: first, through memory pressure as described previously, and second, via user interaction which we use to open a window and load pages into.

Figure 5.11 illustrates the results of this experiment in which the victim's credentials are automatically filled into the page (top) and we successfully recover them (bottom). In addition, Figure 5.12 illustrates the results of a second experiment in which we open a webpage listing which blogs are owned by the victim (top) and successfully recover this list of blogs (bottom) – this list of blogs is ordinarily only accessible to the user. We achieve similar results with either consolidation method.

5.2.5 Exploiting Unintended Content Uploads

Until now, our attacks have assumed that the attacker's webpage resides directly on the domain where the target content was originally uploaded. We now relax this assumption, demonstrating that content uploaded to one domain is sometimes silently transferred to another. This behaviour enables Spook.js to recover the content even when served from a different domain.

Google Sites. As an illustrative case, we examine Google Sites, which allows users to create personal webpages and embed HTML containing JavaScript under *sites.google.com*. Google Sites then executes this user-supplied code within a sandboxed iframe, served from *googleusercontent.com*. As we cannot gain a presence on the *google.com* directly we cannot directly target other *google.com* pages.

However, we observe that Google hosts more than just personal webpages on the content domain. Specifically, Google appears to use the content domain as a general-purpose storage location for user content, automatically uploading email attachments, images, and thumbnails for Google Drive documents.

Google Photos. Focusing on Google Photos, we discover that all images uploaded or automatically synchronised to the service are hosted on the content domain. When users view images through *photos.google.com*, the page loads the images from the content domain using `img` tags. When displayed in this manner, these images are not consolidated with pages from the content domain.

Nevertheless, consolidation can occur if the user chooses alternative methods of viewing the image. For example, if the user opens the image in a new tab, the image loads directly from the content domain. Similarly, accessing the image through a shared link, or a QR code also renders the image directly from the content domain. In all cases, the image will be eligible for consolidation with other pages.

Attack Setup and Results. We create a page on *sites.google.com* that serves Spook.js, which we open in one tab. In a second tab, we open an image uploaded to the victim's private Google Workspace. We then trigger consolidation through memory pressure and recover the image (Figure 5.13).

5.3 Exploiting Malicious Extensions

Moving beyond the security implications of website consolidation, this section explores the security risks associated with the consolidation of Chrome extensions. At a high level, Chrome allows users to install JavaScript-based extensions that modify the browser's default behaviour, such as by blocking ads, applying themes to websites, and managing passwords.

Extension Permissions. To support this functionality, Chrome employs a permissions model that grants extensions capabilities beyond those available to regular JavaScript code executed by websites. To protect these privileged capabilities from both websites and less-privileged extensions, it is essential that Chrome correctly isolates extensions from each other and from web content.

The LastPass Extension. To demonstrate the security implications of Chrome extension consolidation, we examine the LastPass Chrome extension. When a user logs into the LastPass extension, it retrieves an encrypted vault of passwords from LastPass's cloud service and decrypts it using a key derived from the user's master password (LastPass). Our empirical analysis shows that while passwords are decrypted only when needed for autofill, LastPass retains all associated usernames in plaintext in memory.

Attack Setup and Results. We conduct the experiment using Chrome with the LastPass version 4.69.0 extension and sign into our LastPass account. We also port Spook.js into a malicious Chrome extension that requests no permissions and install it on the same system. Then, we consolidate the two extensions into the same process using memory pressure. Finally, we use LastPass to log in to any website, which triggers LastPass to decrypt and populate the website's credentials. Since Spook.js runs in the same process as the LastPass extension, we can access all of its memory, including decrypted credentials (Figure 5.14 top) and the victim's master password (Figure 5.14 bottom).

5.4 Attacking Additional Browsers

We investigate Spook.js on Microsoft Edge and Brave. Edge is the default browser on Windows 10, holding about 5% of the desktop market share (Kinsta), while Brave is a popular privacy-focused browser designed to block ads and trackers (Brave). As both

browsers are built on Chromium, they inherit its site isolation implementation, policies, and associated limitations.

We experimentally confirm that the consolidation techniques are effective in both browsers. Table 5.2 shows that Spook.js achieves leakage rates on Microsoft Edge and Brave comparable to those obtained on Chrome.

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

TABLE 5.2: **Spook.js Performance on Brave and Edge.**

Finally, we test the experimental implementation of site isolation on Firefox (Mozilla Foundation, 2022) using Firefox Nightly version 89.0a1 (build date: 12 April 2021). Similar to Chrome, Firefox Nightly 89.0a1 exhibits consolidation via tab pressure and window.open. However, owing to significant differences in the JavaScript engine, we leave a Firefox port of Spook.js to future work.

5.5 Countermeasures

Separating User JavaScript. Spook.js depends on consolidating two endpoints of the same website into a single process – one executing attacker-controlled JavaScript and the other containing sensitive data. Website operators can protect users from Spook.js by serving these endpoints from different domains. Although this approach is commonly used to separate user content from operator content, it falls short when user-provided JavaScript is served from the same domain as other sensitive user data. We propose extending this strategy by serving user-provided JavaScript from one domain while hosting all other user-provided content on a separate domain. This countermeasure can be readily implemented by website operators to protect users from JavaScript-based attacks such as Spook.js.

Origin Isolation. Browser vendors could align the definition of security domains used in site isolation with those used by the rest of the web. A straightforward approach is to consider the entire domain name for site isolation rather than relying on the eTLD+1.

However, this origin-based isolation may break a significant number of websites, as 13.4% of page loads modify their origin via `document.domain` (Reis et al., 2019).

The Public Suffix List (PSL). Maintained by Mozilla, the PSL (Mozilla Foundation, 2020) is a list of Effective Top Level Domain Names, domain names under which users can directly register subdomains. Examples of entries include *github.io* and, thanks to this work, *bitbucket.io* (Atlassian, 2021). We recommend web services that allow users to register subdomains to add their domains to the PSL to mitigate the risk of related-domain attacks such as Spook.js. In particular, our attacks on *bitbucket.io* and *tumblr.com* were only possible because both domains were absent from the PSL. Notably, Tumblr's absence from the PSL was previously reported by Squarcina et al. (2021).

Strict Extension Isolation. As a result of our findings, Google deployed strict extension isolation (Chromium Project, 2021a). This feature prevents consolidation between extensions and has been enabled in Chrome 92 and later.

Speculation Hardening. At a high level, Spook.js is a type of confusion attack that exploits transient execution beyond an object's type check. A simple, though incomplete, countermeasure is to prevent such transient execution by inserting an `lfence` instruction immediately after all type checks.

A similar technique is pointer poisoning (Pizlo, 2018), which masks each data pointer with a random constant unique to the object's type. This technique does not prevent misprediction of the type check, but will poison the resulting pointer and prevent exploitable transient execution.

5.6 Limitations

Targets. To execute Spook.js, the attacker must upload JavaScript onto the target website's domain. These attacks are known as related-domain attacks (Squarcina et al., 2021; Bortz et al., 2011). Although the chapter has illustrated numerous attack scenarios in which Spook.js can be applied, it cannot be applied across unrelated domains.

Architectures. Due to the non-inclusive cache architecture of AMD Zen processors, our LLC eviction strategy fails on AMD systems. We leave the task of demonstrating Spook.js on non-inclusive caches, such as AMD Zen microarchitectures, to future work.

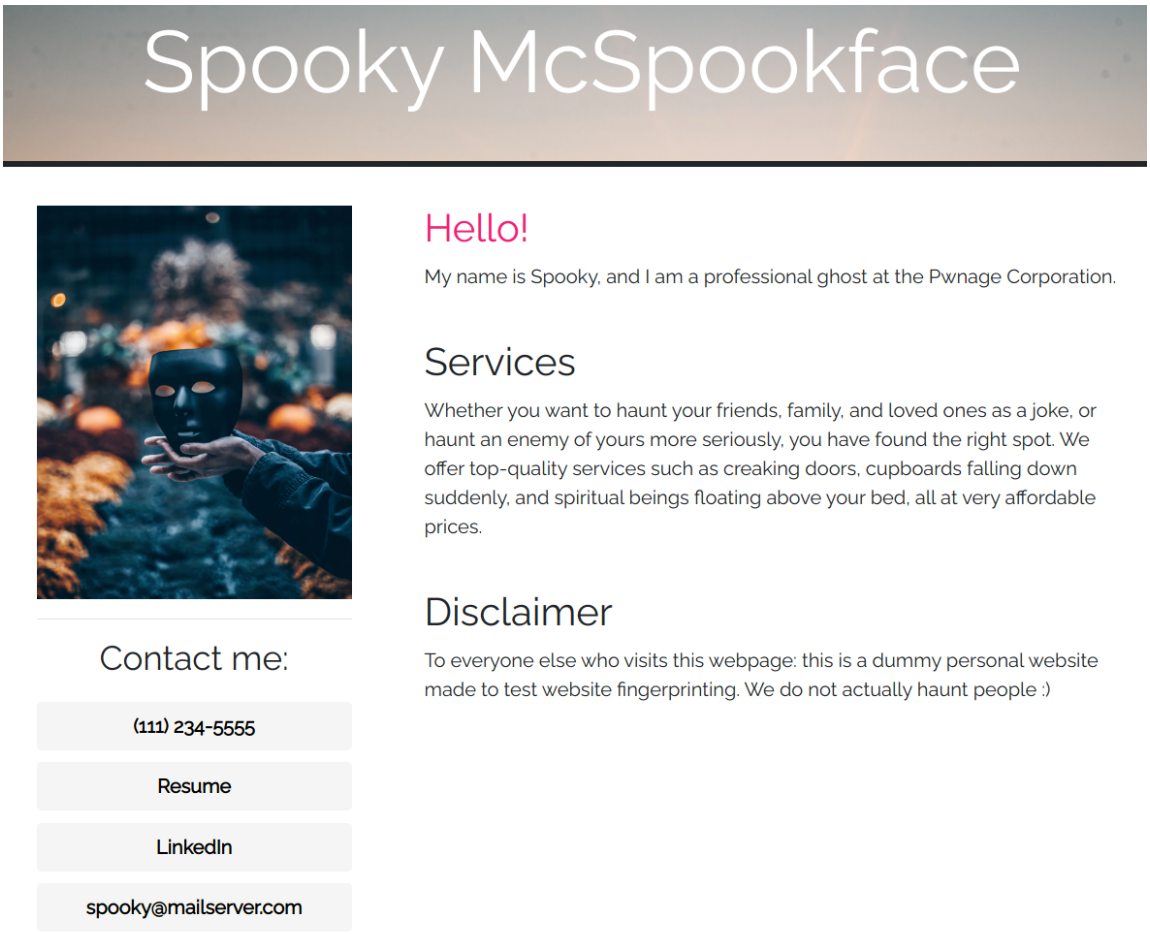
Firefox. Similarly to Chrome, Firefox's site isolation implementation also consolidates websites based on the same-site policy. Although we successfully demonstrated consolidation on Firefox, its JavaScript engine differs significantly from Chrome's. We leave the task of demonstrating Spook.js on firefox to future work.

5.7 Conclusion

In this chapter, we present Spook.js, a novel transient-execution attack capable of recovering sensitive information from other websites despite Chrome's site isolation countermeasures. Similar to other related-domain attacks (Bortz et al., 2011; Squarcina et al., 2021), Spook.js exploits the gap between the policy used by the countermeasure (the same-site policy) and the policy used by the broader web ecosystem (the same-origin policy). We demonstrate several realistic attack scenarios in which we extract sensitive data, including usernames and passwords, from multiple popular web services, browser extensions, and the browser itself. Furthermore, we show that Spook.js can be executed on Chromium-based browsers, such as Brave and Edge, and that Firefox suffers from the same core vulnerability as Chrome.

In response to our work, Google has deployed strict extension isolation, and several popular web services have added themselves to the PSL. While these countermeasures are not as comprehensive as origin isolation or speculation hardening, they were deployed rapidly and with zero impact on compatibility while mitigating the threat posed by Spook.js. Moreover, Google introduced support for configuring how Chromium isolates websites, including an option to enable origin isolation (Chromium Project, 2021b,a).

The next chapter departs from the focus on web browsers while continuing the broader theme of evaluating countermeasures by mounting attacks.




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

FIGURE 5.3: **Results – Bitbucket – Contents:**
(Top) Example of a victim webpage. (Bottom) Leakage of parts of the victim webpage's text. Bitbucket has since mitigated Spook.js.

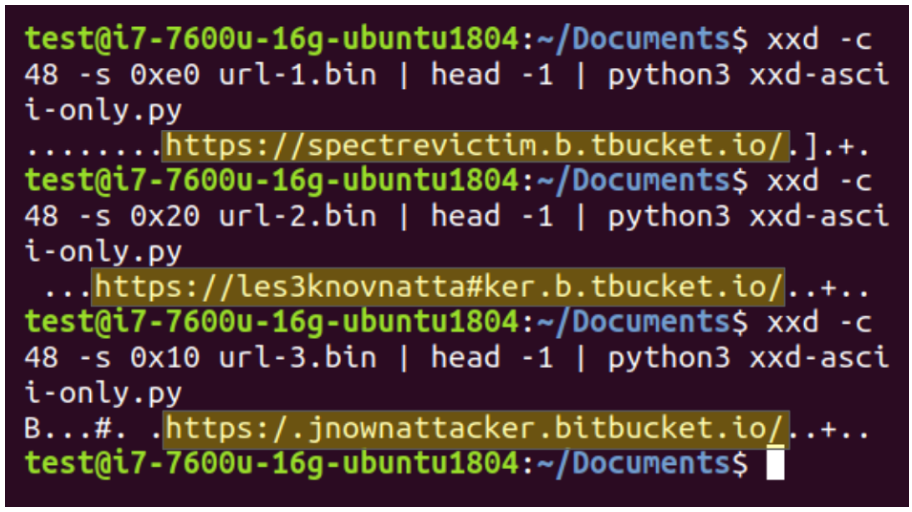


FIGURE 5.4: **Results – Bitbucket – Open Subdomains:** Leakage of currently open *bitbucket.io* subdomains. Parts corresponding to the URLs have been highlighted. Bitbucket has since mitigated Spook.js.

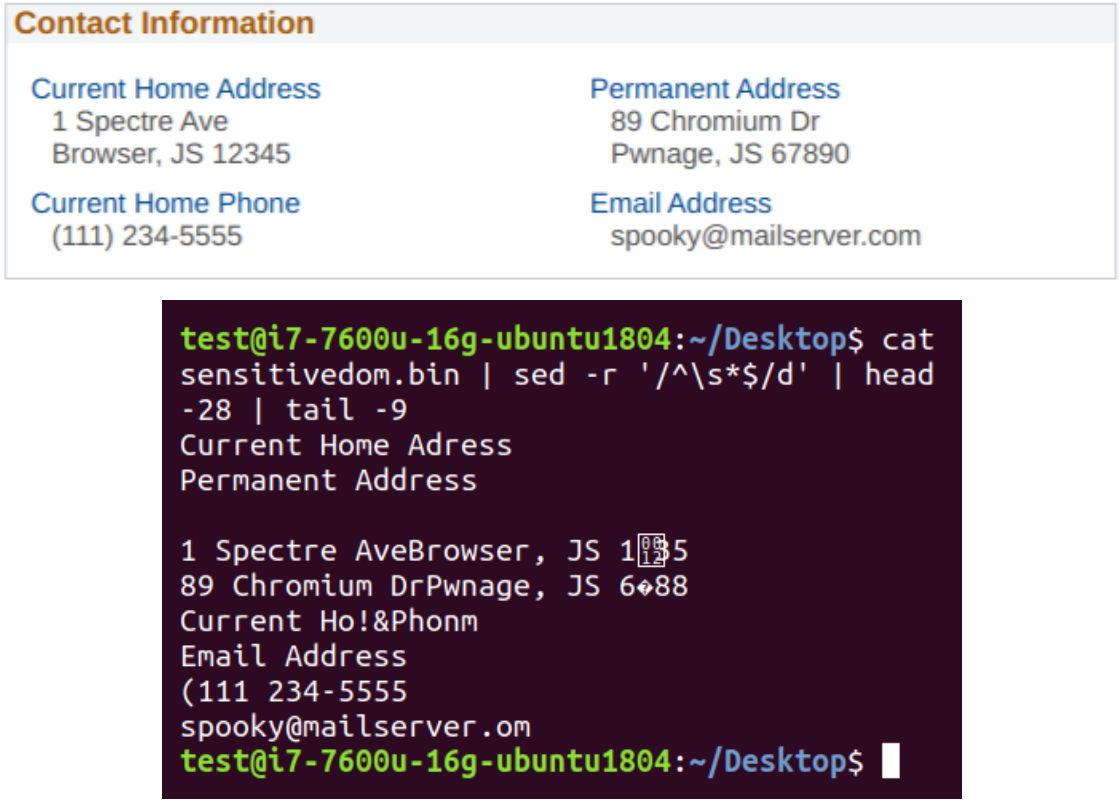


FIGURE 5.5: **Results – University – Contact Information:** (Top) Contact information page displayed of the university website, edited to show anonymized information. (Bottom) Leakage of contact information using Spook.js.

Direct Deposit Details

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

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

FIGURE 5.6: **Results – University – Bank Details:**
(Top) The direct deposit settings page of the university website, edited to show anonymized information. (Bottom) Leakage of bank account and routing number.

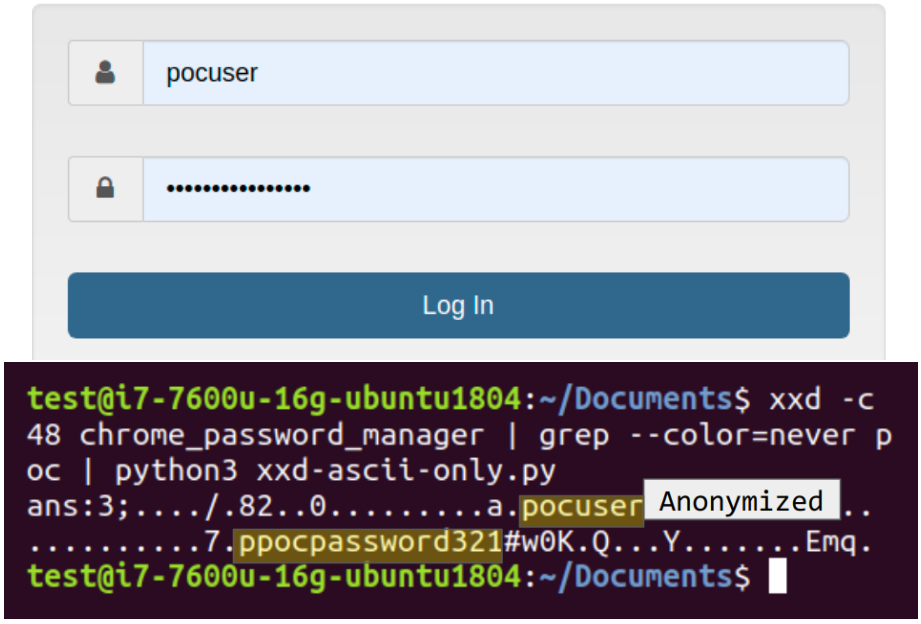
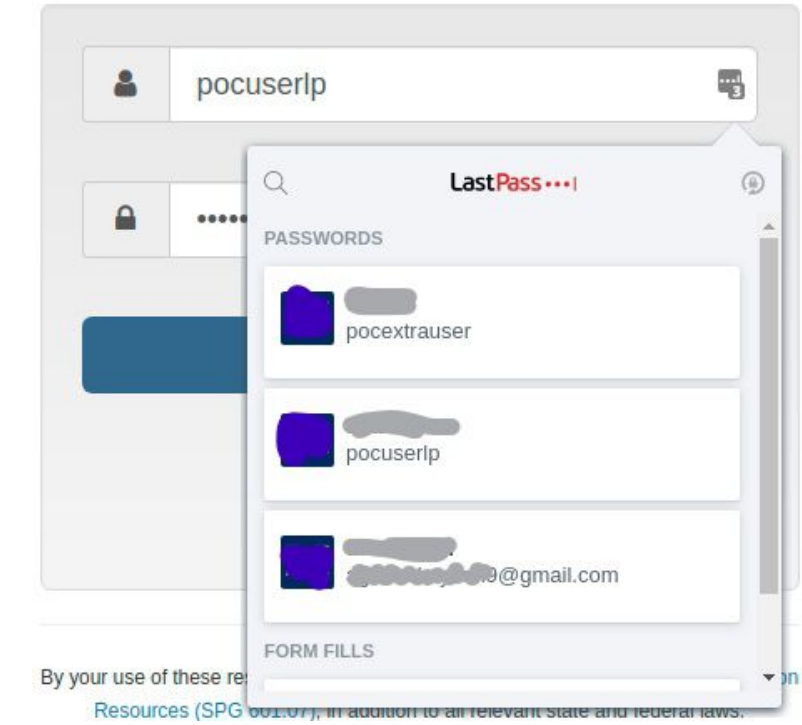


FIGURE 5.7: **Results – Chrome Password Manager – Passwords:** (Top) Credential autofill by Chrome's password manager into the university's login page. (Bottom) Leaked credentials using Spook.js.




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

FIGURE 5.8: **Results – LastPass – Passwords:**
(Top) Multiple accounts managed by LastPass. (Bottom) Using Spook.js to leak the list of associated accounts.

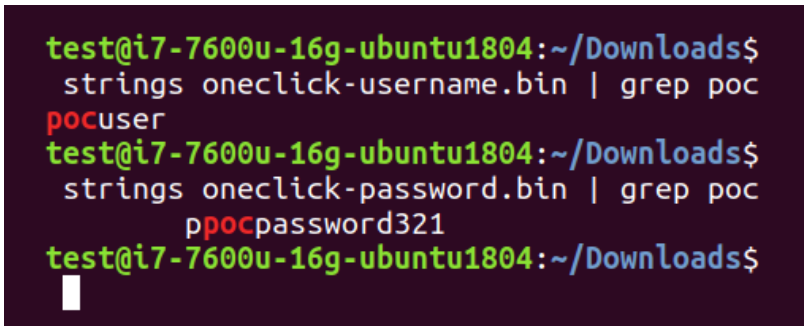


FIGURE 5.9: **Results – Hidden Frame:**
Extracted credentials from a hidden frame embedded in the page.

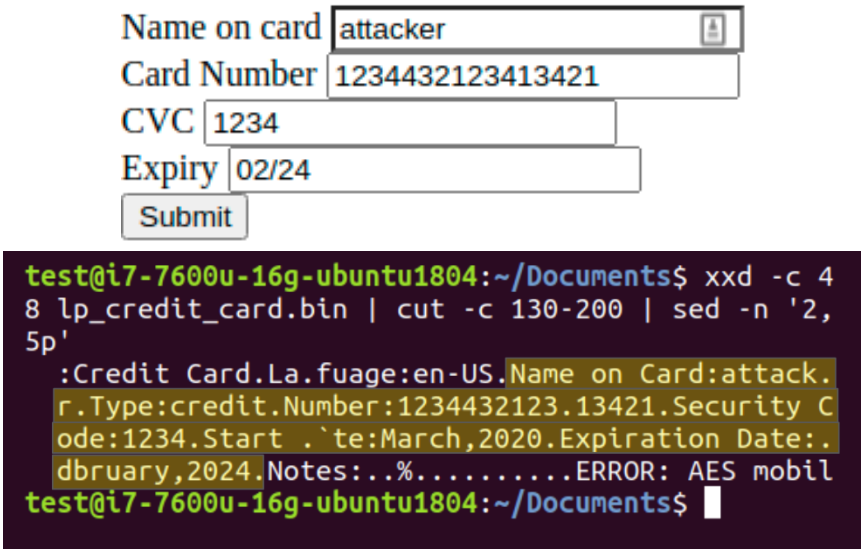
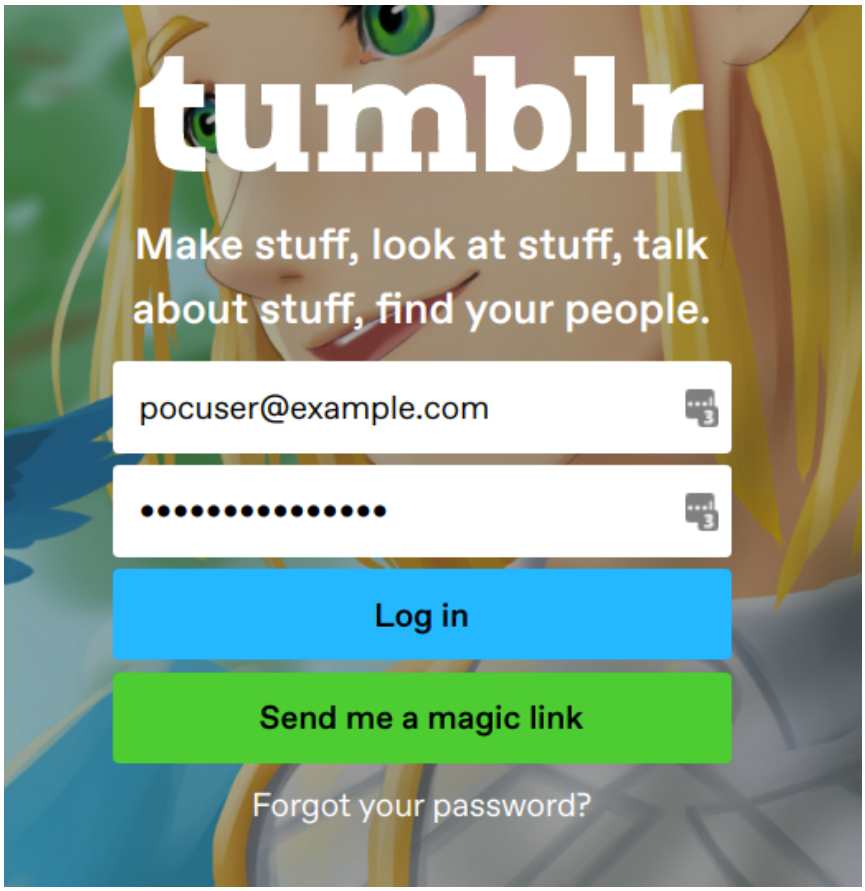
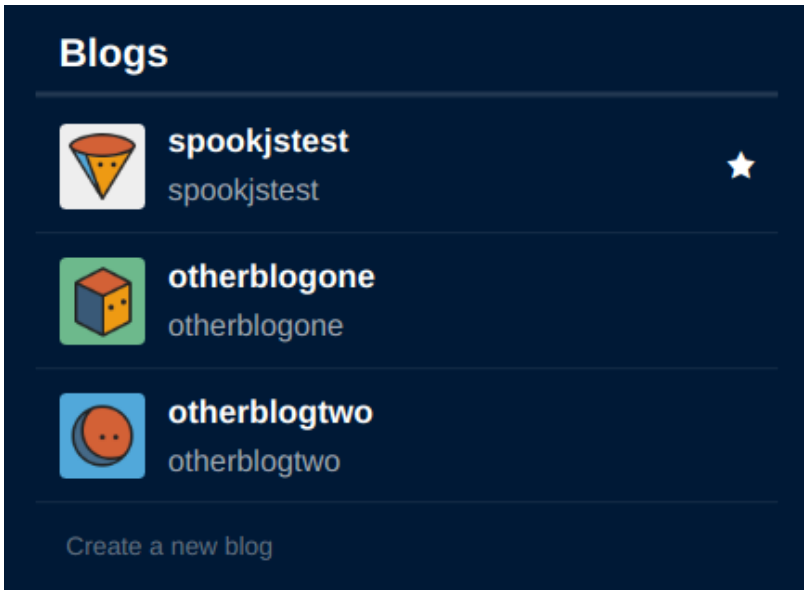


FIGURE 5.10: **Results – LastPass – Credit Cards:**
(Top) Credit card information populated by LastPass. (Bottom) Using Spook.js to leak credit card information.




```
test@i7-7600u-16g-ubuntu1804:~/Documents$ xxd -c 64 tumblr-crede
ntials.bin | head -3 | tail -2 | ./xxd-ascii-only.py
.........OT.pocpassword321.........*.........eitcom.........
.............pocuser@example.com.............n.o.n.e..*..0.K..*..
test@i7-7600u-16g-ubuntu1804:~/Documents$
```

FIGURE 5.11: **Results – Tumblr – Password:**
(Top) Tumblr's login page with credentials autofilled by LastPass. (Bottom) Recovered username and password.




```
test@i7-7600u-16g-ubuntu1804:~/Documents$ xxd -c 64 tumblr-allbl
ogs.bin | head -5 | tail -4 | python3 xxd-ascii-only.py
test&quot;,&quot;title&quot;:&quot;spookjstest&quot;},{&quot;nam
e&quot;:&quot;otherblogo.e&quot;,&quot;title&quot;:&quot;otherbl
2cD...@%...`....Db........}...................w.......{.........
tle&.uot;:&quot;othErblogtwo&qu.t;..,f.AO.`reca0tcha"p.ht;:&quot
test@i7-7600u-16g-ubuntu1804:~/Documents$
```

FIGURE 5.12: **Results – Tumblr – Open Subdomains:** (Top) The list of all blogs owned by an account, which is visible only to the owner of the account. (Bottom) Recovered list with blog names in highlights.

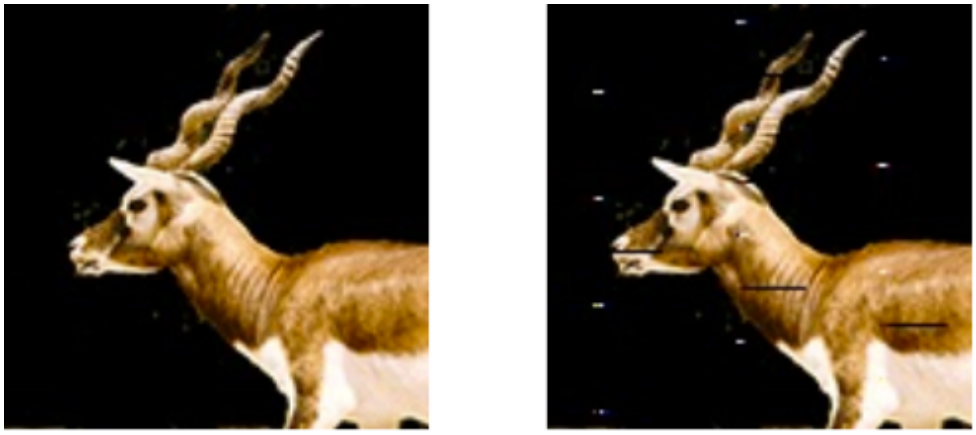


FIGURE 5.13: **Results – Google Photos:**
(Left) An image of an antelope uploaded to Google Photos. (Right) A reconstructed image from the leaked data.

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

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

FIGURE 5.14: **Results – Lastpass – Master Password:**
(Top) Recovered login credential of wikipedia.org populated by LastPass. (Bottom) Recovered master password of LastPass' vault (originally `LastPassPassword1#`).

Chapter 6

Security Type Systems and Transient Execution

As demonstrated in previous chapters, the subtle behaviours of modern processors can be exploited to compromise software security. Fortunately, various techniques have been proposed to mitigate such threats. One prominent approach involves modifying programs to ensure identical microarchitectural behaviour regardless of input. Known as constant-time programming, this method makes microarchitectural observations uninformative to potential adversaries by ensuring these observations remain consistent across all program inputs.

Unfortunately, developing constant-time software presents significant challenges (Jancar et al., 2022). To help overcome these difficulties, several tools have been created to assist in developing constant-time software. These tools can verify whether a program is constant-time and often automatically transform portions of the program into constant-time variants.

The FaCT compiler (Cauligi et al., 2019) is an example of such a tool. It employs a constrained information-flow security type system that ensures secret-typed variables are never used in ways that affect the program's observable behaviour. While these systems are effective, they can be overly restrictive. For example, they prohibit operations such as network transmission or file storage, even when these operations are safe to perform. Moreover, because the results of operations involving secret variables must be assigned to secret variables, this restriction can rapidly propagate throughout the program.

The standard solution to this problem is to introduce a `declassify` operation. This operation serves as an annotation inserted by the developer to assert that a value stored

in a secret variable can be safely stored in a public variable, enabling its use in otherwise forbidden operations. While it can shown that introducing declassify operations is safe under sequential execution, their safety under out-of-order execution remains less clear.

```
uint8_t otp_and_decode(
  secret uint8_t m,
  secret uint8_t k
) {
  secret uint8_t c = m;
  for (uint8_t i = 0; i < 8; i++) {
    c ^= k & (1 << i);
  }

  public uint8_t d = declassify(c);
  return decode[d];
}
```

LISTING 6.1: **One-Time Pad Example:**
One-time pad into table-based decoder. Skipping the for loop (due to misspeculation) directly leaks the secret `m`.

Consider the program in Listing 6.1. In this example, a secret message `m` is encrypted using a bitwise one-time pad, and the resulting ciphertext `c` is passed into a table-based decoder. Because the ciphertext depends on the secret, `c` is classified as secret and must be declassified before it can be decoded. With the assumption that the one-time pad is uniformly distributed, the program does not leak `m` under normal execution.

However, if the loop were not executed at all, then the program would trivially leak `m` via `c`. While such behaviour is impossible under the sequential execution model, since the loop must be executed before the declassify operation can be executed, on real hardware an adversary with control over branch prediction could trick the processor into skipping the loop body entirely.

The remainder of this chapter explores the interaction between declassify operations and out-of-order execution, culminating in a PoC attack targeting several implementations

of the AES encryption algorithm. Unlike prior approaches which focus on enabling adversaries to leak data from unintended locations, such as out-of-bounds memory accesses or type-confused structures, this attack leaks values from the intended location but at an unintended time.

The rest of this chapter provides further background for AES, describes how the PoC attack extracts partial ciphertexts from AES, then ends with a demonstration of the attack on the OpenSSL implementation of AES.

6.1 AES Background

The Advanced Encryption Standard (AES) is a symmetric block cipher that operates on 128-bit blocks and supports key sizes of 128, 192, or 256 bits. AES employs a substitution-permutation network structure, executing multiple rounds of transformations on the plaintext (input) to produce the final encrypted ciphertext (output). The number of rounds depends on the key size, with 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys.

AES Round Overview. The 128-bit AES state is represented as a 4×4 byte matrix and is transformed in each round through four operations: `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`. In the final round the `MixColumns` operation is omitted.

The `SubBytes` operation replaces each byte in the state by another byte using a predefined substitution table (S-box). `ShiftRows` circularly rotates the bytes in row i to left by i positions. `MixColumns` applies a linear transformation to each column, interpreting the bytes as elements of $\mathrm{GF}(2^8)$. Finally, `AddRoundKey` performs bitwise exclusive OR (XOR) between the state and a round-specific key derived from the original key.

AES-NI. The Advanced Encryption Standard New Instructions (AES-NI) is an extension of the x86-64 instruction set that implements the core operations of AES encryption in hardware. It provides improved performance and enhanced security compared with software-based implementations. For AES encryption, AES-NI includes two primary instructions: `AESENC`, which performs a full AES round, and `AESENCLAST`, which performs the final round.

6.2 PoC Attack

The core idea behind the attack is to trick the processor into applying an incorrect number of AES rounds to the plaintext. We do this by training the branch predictor to mispredict branches that control the number of rounds. Listing 6.2 shows a vulnerable implementation of AES. `key` holds all of the round-specific keys along with the number of rounds to apply to the plaintext. The goal of the attack is to control the branches on Lines 18 and 21 to control the number of rounds applied, then leak and analyse the resulting improperly encrypted ciphertext to recover the key.

Step 1: Branch Predictor Training. Listing 6.3 presents the pseudocode for the attack. Lines 2 and 3 create two AES keys, one is used for training and the other represents a 'secret' key that is a stand-in for a key that the attacker does not have access to. Lines 5–7 repeatedly calls `encrypt` using the training key. Because the training key is a 128-bit key, `encrypt` will apply 10 rounds to the plaintext.

In practice, attackers may employ alternative techniques to train the processor, such as exploiting aliasing within the branch prediction unit. For the purposes of this PoC, we use the simple technique of reusing the encryption algorithm for training.

Step 2: Triggering Misspeculation. After training the processor, Line 9 flushes `secret_key.rounds` from the cache. This field controls the number of rounds to apply to the plaintext. Flushing it from the cache delays evaluation of branch conditions within `encrypt` until the field is returned from memory. Then Line 10 calls `encrypt` with the secret key.

As the secret key is a 192-bit key, `encrypt` should apply 12 rounds to the plaintext. However, because of the previous training step, the processor incorrectly applies 10 rounds to the plaintext. The branch that controls the number of rounds (Listing 6.2 Line 18) is predicted as not taken, when it should be taken.

Moreover, because `key.rounds` has been flushed from the cache, the speculation window is large enough for control flow to leave `encrypt` and return back into the attack where Line 12 can leak the resulting reduced-round ciphertext via a side channel.

Step 3: Recovering the Reduced-Round Ciphertext. Eventually, the processor will detect the misprediction and squash the resulting incorrect execution. It restarts execution

back at the branch that controls the number of rounds (Listing 6.2 Line 18) and applies the correct number of rounds then returns back into the attack.

Line 12 is executed again, leaking the resulting correctly encrypted ciphertext. This is simply an artefact of explanation, this is not a necessary step of the attack. In a real attack, the attacker could construct an attack which only leaks the reduced-round ciphertext.

Finally, Line 14 recovers the leaked reduced-round ciphertext using a side channel. We use the Flush+Reload technique (Yarom and Falkner, 2014) as the side-channel technique for simplicity. Several other side channels have been demonstrated in the context of transient-execution attacks (Bhattacharyya et al., 2019; Google, 2021; Amos et al., 2019; Ren et al., 2021), so the choice of channel is not limited to Flush+Reload.

Attack Accuracy. We test two implementations, both written in FaCT: one compiled with the default LLVM back-end and the other with LLVM configured to enable Speculative Load Hardening (SLH). We repeat the attack 1,000 times against each victim, recording whether the reduced-round ciphertext is correctly recovered each time. On average, the attack succeeds with a probability exceeding 95%, regardless of the victim.

OpenSSL Implementations. We further demonstrate leakage from two AES implementations provided by OpenSSL.

The first implementation uses T-tables to perform the round function and follows the same general structure as Listing 6.2 but employs precomputed tables for the round transformations. Although T-table implementations are known to be vulnerable to cache attacks (Osvik et al., 2006), our PoC does not exploit this particular vulnerability; instead, it uses the same strategy described earlier. The PoC succeeds as expected when SLH is disabled, but enabling SLH prevents the leak by poisoning the table accesses executed in the final round.

The second implementation uses AES-NI instructions and is written in x86-64 assembly. The implementation uses `AESENC` within a loop and invokes `AESENCLAST` for the final round. The number of iterations of the loop is determined by `key.rounds`. Because the loop can terminate after any number of iterations, we train the loop to terminate after a single iterations, resulting in a two-round encryption (the last round is unconditionally applied after the loop). The PoC succeeds when SLH is disabled. Since LLVM cannot apply SLH to assembly code, we do not test this implementation with SLH enabled.

6.3 Conclusion

This chapter demonstrated a proof-of-concept attack on declassification. Specifically, the gap between the sequential execution model used by tooling and the out-of-order execution model used by hardware enables adversaries to leak sensitive data from otherwise protected code.

Fortunately, several factors limit the practicality of these attacks. While intra-process isolation remains an active research area (Vahldiek-Oberwagner et al., 2019; Kirth et al., 2022; Voulimeneas et al., 2022), some high-profile applications appear to be moving in the opposite direction (Reis et al., 2019). Moreover, it is unclear whether such an attack could be performed across process boundaries.

The accompanying paper (Shivakumar et al., 2023) provides the remaining context, full theoretical analysis, methods to recover the key, and proposed countermeasures to mitigate this issue. This chapter contains only the work that I contributed to the paper.

```
function encrypt(
  AES_STATE plaintext,
  AES_STATE ciphertext,
  AES_KEY key
) {
  AES_STATE* key = key.round_keys

  AES_STATE state = xor(plaintext, *key++)
  state = AESENC(state, *key++)
  state = AESENC(state, *key++)
  state = AESENC(state, *key++)
  state = AESENC(state, *key++)
  state = AESENC(state, *key++)
  state = AESENC(state, *key++)
  state = AESENC(state, *key++)
  state = AESENC(state, *key++)
  state = AESENC(state, *key++)
  if (key.rounds > 10) {
      state = AESENC(state, *key++)
      state = AESENC(state, *key++)
      if (key.rounds > 12) {
          state = AESENC(state, *key++)
          state = AESENC(state, *key++)
      }
  }
  return AESENCLAST(state, *key++)
}
```

LISTING 6.2: **Protected AES Implementation:** Implementation of unrolled AES encryption. The `aes_round` and `aes_final_round` functions are compiler intrinsics that map to the `AESENC` and `AESENCLAST` x86 instructions respectively.

```
function attack() {
  training_key = create_aes128_key();
  secret_key = create_aes192_key();

  for (int i = 0; i < 127; i++) {
    encrypt(plaintext, training_key);
  }

  flush(secret_key.rounds);
  ciphertext = encrypt(plaintext, secret_key);

  sidechannel_send(ciphertext);

  return sidechannel_recv();
}
```

LISTING 6.3: **Pseudocode for AES Attack:**
Pseudocode of our attack on AES. For clarity, we show training and victim execution as separate steps. In practice, our code does both of these steps in the same loop, using constant-time select to switch between inputs.

Chapter 7

Conclusion

This thesis examines the effectiveness of deployed microarchitectural side-channel countermeasures.

Chapter 3 investigates the root causes of leakage in coarse-grained website fingerprinting attacks. Chapter 4 introduces Pixel Thief– a practical cache-based pixel-stealing attack that recovers pixels faster than any previous timing-based approach. Chapter 5 presents Spook.js– a practical transient execution attack capable of leaking secrets despite the presence of countermeasures. Finally, Chapter 6 describes a PoC demonstrating how transient execution can subtly undermine security type systems.

These attacks demonstrate that at the time of writing, current countermeasures are insufficient to fully protect users from microarchitectural threats. However, this does not imply that the countermeasures have been ineffective. On the contrary, the countermeasures deployed by browser vendors have significantly limited the precision and impact of the attacks presented in this thesis, making them less potent than they might otherwise have been.

For example, Pixel Thief bypasses cross-origin isolation but only by shifting to an adversarial model that requires victim interaction. Similarly, Spook.js circumvents mitigations for transient-execution attacks but only under a related-domain adversarial model. These shifts in the adversarial model are a direct consequence of previously deployed countermeasures and reflect a weakening of the adversary's capabilities to mount effective attacks – a trend that appears to continue with the countermeasures deployed to mitigate Pixel Thief and Spook.js.

In response to Spook.js, Google deployed several targeted mitigations into their

Chrome browser. The browser now correctly isolates extensions into their own processes and provides configurable options for controlling how websites are isolated across processes. However, Chrome still defaults to isolating at the site level, so we recommend, as an interim measure, that operators of vulnerable websites add their domains to the PSL. The PSL changes how the same-site policy is applied, so that subdomains are treated as entirely separate websites. Browser countermeasures that use the same-site policy, including site isolation, will be applied to these subdomains as the operators intended.

Recently deployed cookie isolation features such as Total Cookie Protection and Intelligent Tracking Prevention also affect Pixel Thief and Spook.js. These features aim to prevent user tracking by isolating cookies based on whether a website is loaded normally or embedded within an iframe on a third-party site. A side effect of these features is that iframe-based attacks are served fresh instances of websites without any cookies. Without any cookies, the iframe will likely not contain any of the victim's sensitive information, defeating iframe-based attacks.

Despite this ongoing progress, the state of countermeasures remains far from ideal. Firstly, Browser vendors must balance security with other competing concerns, including compatibility and performance, and may not be able to implement strong countermeasures. For instance, browsers still allow cross-origin application of filters, despite their security risk, because existing websites rely on this capability. Similarly, browsers have begun to adopt website isolation, but continue to use the same-site policy instead of the same-origin policy for compatibility and performance reasons.

Moreover, the implementations of these countermeasures may themselves be flawed. For example, Kim et al. (2023) uncovered a flaw in Safari that allowed an adversary to consolidate arbitrary websites into the same process undermining its isolation countermeasures entirely. Apple has since patched this vulnerability; however it raises questions about whether similar vulnerabilities could exist.

Finally, countermeasures are incomplete. Consider the coarse-grained website-fingerprinting attacks described in Chapter 3. It is simply unclear how a viable countermeasure could ever mitigate such attacks.

Despite these limitations in countermeasures, there is room for optimism in this space. The existing countermeasures were very effective at limiting the capabilities of Pixel Thief and Spook.js, and the newly deployed countermeasures only further limit their capability. Importantly, these countermeasures have been deployed and are now in active use.

I urge the reader not to be discouraged by these limitations, but to take them as a call to action to improve the security of the web. Finally, I would like to end the thesis by thanking you, the reader, for your time.

– Cheers, Sioli

Bibliography

Onur Acıiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *FDTC*, pages 80–91, 2007. doi: 10.1109/fdtc.2007.16.

Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *CT-RSA*, pages 225–242, 2006. doi: 10.1007/11967668_15.

Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *IMACC*, pages 185–203, 2007. doi: 10.1007/978-3-540-77272-9_12.

Alejandro Cabrera Aldaya and Billy Bob Brumley. HyperDegrade: From GHz to MHz effective CPU frequencies. In *USENIX Security*, pages 2801–2818, 2022.

Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *IEEE SP*, pages 870–887, 2019. doi: 10.1109/sp.2019.00066.

Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *ACSAC*, pages 422–435, 2016. doi: 10.1145/2991079.2991084.

Ben Amos, Niv Gilboa, and Arbel Levy. Spectre without shared memory. In *SAC*, pages 1944–1951, 2019. doi: 10.1145/3297280.3297470.

Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *IEEE SP*, pages 623–639, 2015. doi: 10.1145/3243734.3243766.

Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards verified, constant-time floating point operations. In *CCS*, pages 1369–1382, 2018. doi: 10.1145/3243734.3243766.

Atlassian. Spook.js: speculative execution resulting in cross-domain browser information leakage, 2021. URL *https://community.atlassian.com/t5/Trust-Security-Articles/Spook-js-speculative-execution-resulting-in-cross-domain-browser/ba-p/1799650*.

Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *CCS*, pages 1267–1279, 2014. doi: 10.1145/2660267.2660283.

Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *CCS*, pages 785–800, 2019. doi: doi/10.1145/3319535.3363194.

Andrew Bortz, Adam Barth, and Alexei Czeskis. Origin cookies: Session integrity for web applications. *W2SP*, 2011.

Brave. The brave browser. URL *https://brave.com/*.

Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities, 2006.

William J. Buchanan, Scott Helme, and Alan Woodward. Analysis of the adoption of security headers in HTTP. In *IET Information Security*, pages 118–126, 2018. doi: 10.1049/iet-ifs.2016.0621.

Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *CCS*, pages 605–616, 2012. doi: 10.1145/2382196.2382260.

Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo van Bulck, and Yuval Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *CCS*, pages 769–784, 2019. doi: 10.1145/3319535.3363219.

Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. FaCT: A DSL for timing-sensitive computation. In *PLDI*, pages 174–189, 2019. doi: 10.1145/3314221.3314605.

Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. *ACM SIGPLAN Notices*, pages 49–70, 1989. doi: 10.1145/74877.74884.

Chromium Project. SVG filter timing attack, 2013. URL *https://bugs.chromium.org/p/chromium/issues/detail?id=251711*.

Chromium Project. Timing attack on denormalized floating point arithmetic in SVG filters circumvents same-origin policy, 2016a. URL *https://bugs.chromium.org/p/chromium/issues/detail?id=615851*.

Chromium Project. window.performance.now does not support sub-millisecond precision on Windows, 2016b. URL *https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110*.

Chromium Project. Cross-origin pixel reading and history sniffing via SVG filter timing attack, 2017. URL *https://bugs.chromium.org/p/chromium/issues/detail?id=686253*.

Chromium Project. Protecting more with site isolation, 2021a. URL *https://security.googleblog.com/2021/07/protecting-more-with-site-isolation.html*.

Chromium Project. Site isolation, 2021b. URL *https://www.chromium.org/Home/chromium-security/site-isolation/*.

Chromium Project. Software rendering list, 2023. URL *https://chromium.googlesource.com/chromium/src/gpu/+/refs/heads/main/config/software_rendering_list.json*.

Jack Cook, Jules Drean, Jonathan Behrens, and Mengjia Yan. There's always a bigger fish: A clarifying analysis of a machine-learning-assisted side-channel attack. In *ISCA*, pages 204–217, 2022. doi: 10.1145/3470496.3527416.

Patrick Cronin, Xing Gao, Haining Wang, and Chase Cotton. An exploration of ARM system-level cache and GPU side channels. In *ACSAC*, pages 784–795, 2021. doi: 10.1145/3485832.3485902.

Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. Don't mesh around: Side-Channel attacks and mitigations on mesh interconnects. In *USENIX Security*, pages 2857–2874, 2022.

L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL*, pages 297–302, 1984. doi: 10.1145/800017.800542.

Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*, pages 1–13, 2016. doi: 10.1109/micro.2016.7783743.

Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *ACM SIGPLAN Notices*, pages 693–707, 2018. doi: 10.1145/3296957.3173204.

The OWASP Foundation. The OWASP secure headers project. URL *https://owasp.org/www-project-secure-headers/*.

Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the Spectre era. In *CCS*, pages 1871–1885, 2020. doi: 10.1145/3372297.3417289.

Xun Gong, Nikita Borisov, Negar Kiyavash, and Nabil Schear. Website detection using remote traffic analysis. In *PETS*, pages 58–78, 2012. doi: 10.1007/978-3-642-31680-7_4.

Google. Leaky page, 2021. URL *https://leaky.page*.

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

David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *IEEE SP*, pages 490–505, 2011. doi: 10.1109/SP.2011.22.

Berk Gülmezoglu. Xai-based microarchitectural side-channel analysis for website fingerprinting attacks and defenses. *IEEE TDSC*, pages 4039–4051, 2021. doi: 10.1109/tdsc.2021.3117145.

Noam Hadad and Jonathan Afek. Overcoming (some) Spectre browser mitigations, 2018. URL *https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/*.

Jamie Hayes and George Danezis. k-fingerprinting: A robust scalable website fingerprinting technique. In *USENIX Security*, pages 1187–1203, 2016.

Scott Helme. Top 1 million analysis - june 2022. URL *https://scotthelme.co.uk/top-1-million-analysis-june-2022/*.

Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In *CCSW*, pages 31–42, 2009. doi: 10.1145/1655008.1655013.

Andrew Hintz. Fingerprinting websites using traffic analysis. In *PETS*, pages 171–178, 2002. doi: 10.1007/3-540-36467-6_13.

Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP*, pages 21–38, 1991. doi: 10.1007/bfb0057013.

Anxin Huang, Chen Zhu, Dewen Wu, Yi Xie, and Xiapu Luo. An adaptive method for cross-platform browser history sniffing. In *MADWeb*, pages 1–7, 2020. doi: 10.14722/madweb.2020.23006.

Lin-Shung Huang, Alexander Moshchuk, Helen J. Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and defenses. In *USENIX Security*, pages 413–428, 2012.

Intel. Intel 64 and IA-32 architectures software developer's manual. URL *https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html*.

Artur Janc and Lukasz Olejnik. Web browser history detection as a real-world privacy threat. In *ESORICS*, pages 215–231, 2010. doi: 10.1007/978-3-642-15497-3_14.

Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "They're not that hard to mitigate": What cryptographic library developers think about timing attacks. In *IEEE SP*, pages 632–649, 2022. doi: 10.1109/SP46214.2022.9833713.

Rob Jansen, Marc Juarez, Rafa Galvez, Tariq Elahi, and Claudia Diaz. Inside job: Applying traffic analysis to measure tor from within. In *NDSS*, 2018. doi: 10.14722/ndss.2018.23261.

Marc Juarez, Sadia Afroz, Gunes Acar, Claudia Diaz, and Rachel Greenstadt. A critical evaluation of website fingerprinting attacks. In *CCS*, pages 263–274, 2014. doi: 10.1145/2660267.2660368.

David A Kaplan. Optimization and amplification of cache side channel signals, 2023.

Georgios Karopoulos, Dimitris Geneiatakis, and Georgios Kambourakis. Neither good nor bad: A large-scale empirical analysis of HTTP security response headers. In *TrustBus*, pages 83–95, 2021. doi: 10.1007/978-3-030-86586-3_6.

Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. The gates of time: Improving cache attacks with transient execution. In *USENIX Security*, 2023.

Jason Kim, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. iLeakage: Browser-based timerless speculative execution attacks on apple devices. In *CCS*, pages 2038–2052, 2023. doi: 10.1145/3576915.3616611.

Kinsta. Global desktop browser market share. URL *https://kinsta.com/browser-market-share/*.

Vladimir Kiriansky and Carl A. Waldspurger. Speculative buffer overflows: Attacks and defenses, 2018.

Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. PKRU-Safe: Automatically locking down

the heap between safe and unsafe languages. In *EuroSys*, pages 132–142, 2022. doi: 10.1145/3492321.3519582.

Ofek Kirzner and Adam Morrison. An analysis of speculative type confusion vulnerabilities in the wild. In *USENIX Security*, 2021.

Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE SP*, pages 1–19, 2019. doi: 10.1109/SP.2019.00002.

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

Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: Breaking KASLR on the isolated kernel address space using tagged TLBs. In *EuroS&P*, pages 309–321, 2020. doi: 10.1109/eurosp48549.2020.00027.

Robert Kotcher, Yutong Pei, Pranjal Jumde, and Collin Jackson. Cross-origin pixel stealing: Timing attacks using CSS filters. In *CCS*, pages 1055–1062, 2013. doi: 10.1145/2508859.2516712.

Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, pages 613–615, 1973. doi: 10.1145/362375.362389.

Arturs Lavrenovs and F. Jesus Rubio Melon. HTTP security headers analysis of top one million websites. In *CyCon*, pages 345–370, 2018. doi: 10.23919/CYCON.2018.8405025.

Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, pages 707–710, 1966.

Shuai Li, Huajun Guo, and Nicholas Hopper. Measuring information leakage in website fingerprinting attacks and defenses. In *CCS*, pages 1977–1992, 2018. doi: 10.1145/3243734.3243832.

Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike

Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, pages 973–990, 2018.

Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, pages 605–622, 2015. doi: 10.1109/SP.2015.43.

Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, William Robertson, Engin Kirda, and Anil Kurmus. Bypassing memory safety mechanisms through speculative control flow hijacks, 2020.

Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *RAID*, pages 48–65, 2015. doi: 10.1007/978-3-319-26362-5_3.

MDN Contributors. The <fecomponenttransfer> svg filter primitive, a. URL *https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feComponentTransfer*.

MDN Contributors. Privacy and the :visited selector, b. URL *https://developer.mozilla.org/en-US/docs/Web/CSS/Privacy_and_the_:visited_selector*.

MDN Contributors. Cross-origin-embedder-policy, 2024a. URL *https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Embedder-Policy*.

MDN Contributors. Cross-origin-opener-policy, 2024b. URL *https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy*.

Benedikt Meurer. An introduction to speculative optimization in v8, 2017. URL *https://ponyfoo.com/articles/an-introduction-to-speculative-optimization-in-v8*.

Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES*, pages 69–90, 2017. doi: 10.1007/978-3-319-66787-4_4.

Mozilla Bug Tracker. SVG filter timing attack, 2013. URL *https://bugzilla.mozilla.org/show_bug.cgi?id=711043*.

Mozilla Foundation. Public suffix list, 2020. URL *https://publicsuffix.org/*.

Mozilla Foundation. Project Fission, 2022. URL *https://wiki.mozilla.org/Project_Fission*.

Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: GPU side channel attacks are practical. In *CCS*, pages 2139–2153, 2018. doi: 10.1145/3243734.3243831.

Nick Nguyen. The best firefox ever, 2017. URL *https://blog.mozilla.org/en/products/firefox/faster-better-firefox/*.

Keith O'Neal and Scott Yilek. Interactive history sniffing with dynamically-generated QR codes and CSS difference blending. In *SP Workshops*, pages 335–341, 2022. doi: 10.1109/spw54247.2022.9833863.

Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *CCS*, pages 1406–1418, 2015. doi: 10.1145/2810103.2813708.

Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, pages 1–20, 2006. doi: 10.1007/11605805_1.

Rodney Owens and Weichao Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *IPCCC*, pages 1–8, 2011. doi: 10.1109/pccc.2011.6108094.

Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. Lord of the ring(s): Side channel attacks on the CPU On-Chip ring interconnect are practical. In *USENIX Security*, pages 645–662, 2021.

Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *WPES*, pages 103–114, 2011. doi: 10.1145/2046556.2046570.

Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. Website fingerprinting at internet scale. In *NDSS*, 2016. doi: 10.14722/ndss.2016.23477.

Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

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

Filip Pizlo. What Spectre and Meltdown mean for WebKit, 2018. URL *https://webkit.org/blog/8048/what-spectreand-meltdown-mean-for-webkit/*.

Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks. In *CCS*, pages 2906–2920, 2021. doi: 10.1145/3460120.3484816.

Charles Reis, Adam Barth, and Carlos Pizano. Browser security: lessons from google chrome. *Communications of the ACM*, pages 45–49, 2009. doi: 10.1145/1551644.1556050.

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

Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. I see dead μops: Leaking secrets via Intel/AMD micro-op caches. In *ISCA*, pages 361–374, 2021. doi: ISCA52012.2021.00036.

Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. *NDSS*, 2018. doi: 10.14722/ndss.2018.23105.

Gustav Rydstedt, Baptiste Gourdin, Elie Bursztein, and Dan Boneh. Framing attacks on smart phones and dumb routers: Tap-jacking and geo-localization attacks. In *WOOT*, 2010.

Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *Financial Cryptography*, pages 247–267, 2017. doi: 10.1007/978-3-319-70972-7_13.

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

Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. Spectre declassified: Reading from the right place at the wrong time. In *IEEE SP*, pages 1753–1770, 2023. doi: 10.1109/SP46215.2023.10179355.

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

Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses. In *USENIX Security*, pages 2863–2880, 2021.

Michael Smith, Craig Disselkoen, Shravan Narayan, Fraser Brown, and Deian Stefan. Browser history re: visited. In *WOOT*, 2018.

Marco Squarcina, Mauro Tempesta, Lorenzo Veronese, Stefano Calzavara, and Matteo Maffei. Can i take your subdomain? exploring Same-Site attacks in the modern web. In *USENIX Security*, pages 2917–2934, 2021.

statista. Combined desktop and mobile visits to Tumblr.com from May 2019 to January 2021, 2021. URL *https://www.statista.com/statistics/261925/unique-visitors-to-tumblrcom/*.

Paul Stone. Pixel perfect timing attacks with HTML5, 2013.

Hritvik Taneja, Jason Kim, Jie Jeff Xu, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. Hot pixels: Frequency, power, and temperature attacks on GPUs and ARM SoCs. *USENIX Security*, 2023.

The HTTP Archive. The HTTP archive almanac. URL *http://almanac.httparchive.org/en/2022/security*.

Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: secure, efficient in-process isolation with protection keys (MPK). In *USENIX Security*, pages 1221–1238, 2019.

Jo van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *SysTEX@SOSP*, pages 4:1–4:6, 2017. doi: 10.1145/3152701.3152706.

Victor van der Veen and Ben Gras. DramaQueen: Revisiting side channels in DRAM. In *DRAMSec*, 2023.

Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why stopping cache attacks in software is harder than you think. In *USENIX Security*, pages 937–954, 2018.

Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. In *IEEE SP*, pages 39–54, 2019. doi: 10.1109/sp.2019.00042.

Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. CacheQuery: Learning replacement policies from hardware caches. In *PLDI*, pages 519–532, 2020. doi: 10.1145/3385412.3386008.

Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You shall not (by)pass! practical, secure, and fast PKU-based sandboxing. In *EuroSys*, pages 266–282, 2022. doi: 10.1145/3492321.3519560.

Frederick M Waltz and John WV Miller. Efficient algorithm for Gaussian blur using finite-state machines. In *Machine Vision Systems for Inspection and Metrology VII*, pages 334–341, 1998. doi: 10.1117/12.326976.

Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. MeshUp: Stateless cache side-channel attack on CPU mesh. In *IEEE SP*, pages 1506–1524, 2022. doi: 10.1109/SP46214.2022.9833794.

Tao Wang and Ian Goldberg. Improved website fingerprinting on Tor. In *WPES*, pages 201–212, 2013. doi: 10.1145/2517840.2517851.

Yao Wang, Andrew Ferraiuolo, and G Edward Suh. Timing channel protection for a shared memory controller. In *HPCA*, pages 225–236, 2014. doi: 10.1109/hpca.2014.6835934.

Yingchen Wang, Riccardo Paccagnella, Elizabeth He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. Hertzbleed: Turning power side-channel attacks into remote timing attacks on x86. In *USENIX Security*, 2022.

Yingchen Wang, Riccardo Paccagnella, Alan Wandke, Zhao Gang, Grant Garrett-Grossman, Christopher W. Fletcher, David Kohlbrenner, and Hovav Shacham. DVFS frequently leaks secrets: Hertzbleed attacks beyond SIKE, cryptography, and CPU-only data. In *IEEE SP*, 2023. doi: 10.1109/SP46215.2023.10179326.

Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. Leaky DNN: Stealing deep-learning model secret with GPU context-switching side-channel. In *DSN*, pages 125–137, 2020. doi: 10.1109/DSN48063.2020.00031.

Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *IEEE SP*, pages 888–904, 2019. doi: 10.1109/SP.2019.00004.

Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, pages 719–732, 2014.

Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. (m)wait for it: Bridging the gap between microarchitectural and architectural side channels. In *USENIX Security*, 2023.

Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring branch predictors for constructing transient execution trojans. In *ASPLOS*, pages 667–682, 2020. doi: 10.1145/3373376.3378526.

Appendix A

Full Address Calculation

Alongside pointer compression, Chrome's JavaScript engine employs a technique known as value tagging (Sheludko and Solanes, 2020), which enables it to use the same 32-bit space to represent both integers and pointers.

The least significant bit (LSB) is used to differentiate whether the 32-bit value encodes an integer or a pointer to an object. For integers, the LSB is unset, and the remaining 31 bits represent the integer value. For pointers, the LSB is set, and the offset for any access involving the pointer is decremented by one – this can be accomplished because all objects are aligned to a 4-byte multiple. Operations on values then check the least significant bit to determine whether the value is an integer or a pointer, removing the need for indirection in integer operations or the need for an additional type-identifying field.

Because Chrome does not use value tagging for internal fields, Chrome interprets the `external` and `base` fields of `Uint8Array` as a regular 64-bit address and 32-bit integer. In contrast, the fields of `AttackerClass` are value tagged and will be encoded as 31-bit integers by shifting their values one bit to the left. When we trigger type confusion, values that we stored in our `AttackerClass` object will be interpreted differently by Chrome.

We can partially resolve this difference in representation by reversing the effects of the value encoding (shifting every value one bit to the right), but we cannot set the least significant bit of any field. Since two fields are confused with `external` (the 64-bit address), we cannot directly set bits 1 and 33 of the address.

We overcome this challenge by abusing how array indexing adds the `external` and `base` fields to the index. Specifically, we set most of the address bits in `external` and use `base` and the index to induce overflows to set bits 1 and 33 of the address. To set bit 1,

we leave `base` as 0 and set index to 1. To set bit 33, we set `base` to `0xFFFFFFFE` and set index to 2. To set both bits, we set `base` to `0xFFFFFFFE` and set index to 3.