

**CHARACTERIZING MICROARCHITECTURAL SIDE-CHANNEL THREATS
ON THE SECURITY OF WEB BROWSERS**

A Dissertation
Presented to
The Academic Faculty

By

Jason Kim

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Cybersecurity and Privacy
College of Computing

Georgia Institute of Technology

December 2025

© Jason Kim 2025

**CHARACTERIZING MICROARCHITECTURAL SIDE-CHANNEL THREATS
ON THE SECURITY OF WEB BROWSERS**

Thesis committee:

Dr. Daniel Genkin
School of Cybersecurity and Privacy
Georgia Institute of Technology

Dr. Christina Garman
Department of Computer Science
Purdue University

Dr. Taesoo Kim
School of Cybersecurity and Privacy
Georgia Institute of Technology

Dr. Christopher W. Fletcher
Department of Electrical Engineering and
Computer Sciences
University of California, Berkeley

Dr. Paul Pearce
School of Cybersecurity and Privacy
Georgia Institute of Technology

Date approved: November 21, 2025

ACKNOWLEDGMENTS

Here culminates the marathon known as earning a doctoral degree. In retrospect, finishing it would not have been possible without the many people around me who have played the role of pacers, aid stations, and tailwinds. Firstly, I am grateful to my advisor, Daniel Genkin, with whom I got my feet wet in security in Fall 2019 as his Teaching Assistant for Michigan’s EECS 388, and who has guided me into maturing as an independent researcher ever since. It has truly been a great run. I also extend my thanks to my thesis committee – Taesoo Kim, Paul Pearce, Christina Garman, and Chris Fletcher – for the pieces of advice from random moments that have shaped me into who I am today.

The works in this thesis would not have existed without the rigor spent by my co-authors and collaborators, and I thank and acknowledge them by name here: Ayush Agarwal, Jonathan Berger, Jalen Chuang, Ingab Kang, Andrew Kwong, Kevin Leach, Sioli O’Connell, Eyal Ronen, Stephan van Schaik, Hovav Shacham, Hritvik Taneja, Youssef Tobah, Riad Wahby, Walter Wang, Jeff Xu, Yuval Yarom, and Shaked Yehezkel.

Neither would this thesis exist without the friends, former labmates, research cousins, and fellow SCP students who had made life at the lab and at Georgia Tech exciting and guided me through my most arduous moments. Thus, I would like to express my gratitude to Samy Amer, Aranya Banerjee, Abhi Bhaskar, Camille Bossut, Rachel Calder, Boru Chen, Daniel Collins, Robbie Dumitru, Byeongyong Go, Amanda Hsu, Hugo Hue, Nuredin Kamadan, Dhruv Kuchhal, Akshaya Kumar, Brad Morgan, Saideep Narendrula, Anna Raymaker, Zeezoo Ryu, Anish Saxena, Roy Stracovsky, Eric Torres, Kyle Watters, Zahra Yazdani, and Zhiyuan Zhang. I will miss our potlucks, hikes, and gatherings.

Last but not least, I express my deep appreciation for my family – my parents, Irene, Moon, and Youngmi; my siblings, Eubin, Eugene, Jessy, and John; my wife, Angela; my parents-in-law, Yingzi and Yong; and my brother-in-law, Benson – who have raised me, supported me, encouraged this long journey, and ensured that I am on the right path. Mak-

ing this thesis and the works within entailed one and a half years of a long-distance relationship (and subsequently marriage) combined with frequent traveling, and for this I am especially grateful to my better half for believing in me.

TABLE OF CONTENTS

Acknowledgments	iii
List of Tables	x
List of Figures	xi
List of Acronyms	xvi
Summary	xviii
Chapter 1: Introduction	1
1.1 Thesis Organization and Contribution Summary	2
Chapter 2: Background	6
2.1 Sources of Speculative Execution	6
2.2 Side-Channel Attacks on Web Browsers	7
Chapter 3: iLeakage: Browser-based Timerless Speculative Execution Attacks on Apple Devices	9
3.1 Introduction	10
3.1.1 Our Contributions	11
3.1.2 Ethics and Artifact Availability	14
3.2 Background	14

3.2.1	Caches and Cache Attacks	14
3.2.2	Speculative and Out-of-Order Execution	15
3.2.3	Side Channel Attacks on Apple CPUs	15
3.2.4	Side Channel Attacks in Web Browsers	16
3.2.5	Timing Sources on Apple CPUs	17
3.2.6	Timer-Friendly Covert Channels	18
3.3	Threat Model	20
3.4	Microarchitectural Primitives	20
3.4.1	[P_1]: Cache Organization	21
3.4.2	[P_2]: Measuring Speculation Depth	23
3.4.3	[P_3]: Discerning Cache Hits From Misses	27
3.4.4	[P_4]: Constructing Minimal Eviction Sets	30
3.4.5	[P_5]: Testing for L2 Inclusiveness	31
3.4.6	[P_6]: Timerless Spectre Attacks	33
3.5	Attacking the Safari Browser	34
3.5.1	[C_1]: Bypassing Process Isolation	35
3.5.2	[C_2]: Speculative Type Confusion	36
3.5.3	[C_3]: Partial Object Eviction	41
3.5.4	[C_4]: Speculative Suppression	43
3.5.5	End-to-End Attack Evaluation	44
3.6	Weaponizing iLeakage	45
3.7	Discussion	48
3.7.1	Countermeasures for Safari	48

3.7.2	Countermeasures for Websites	49
3.7.3	Limitations	50
3.7.4	Broader Implications	51
3.8	Conclusion	52
Chapter 4: SLAP: Data Speculation Attacks via Load Address Prediction on Apple Silicon		53
4.1	Introduction	54
4.1.1	Our Contributions	55
4.1.2	Responsible Disclosure	57
4.2	Background	58
4.3	Threat Model	60
4.4	Reverse Engineering the LAP	60
4.4.1	Observing Load Data Speculation	61
4.4.2	Confirming Load Address Prediction	64
4.4.3	Confirming Speculative Execution via LAP	67
4.4.4	Spatial Conditions for Speculation	71
4.4.5	Temporal Conditions for Speculation	73
4.4.6	Confirming Instruction Address Tagging	77
4.5	Weaponizing the LAP for Attacks	77
4.5.1	Spectre-LAP	77
4.5.2	Hijacking Control Flow	79
4.5.3	Evaluation and Breaking ASLR	80
4.6	Attacking Safari With the LAP	82

4.6.1	Timer-Resilient Covert Channel	82
4.6.2	SLAP Gadget Overview	83
4.6.3	Exploiting Safari’s Memory Model	84
4.6.4	Reading Data Across Websites	86
4.7	Mitigations	89
4.8	Conclusion	91
Chapter 5: FLOP: Breaking the Apple M3 CPU via False Load Output Predictions		93
5.1	Introduction	93
5.1.1	Our Contributions	94
5.2	Background	97
5.3	Threat Model and Setup	99
5.4	Analysis of the Apple M3 LVP	100
5.4.1	Ascertaining the LVP’s Presence	100
5.4.2	Identifying LVP Activation Criteria	103
5.4.3	Measuring Mispredicted Load Values	107
5.4.4	Inducing Memory Safety Violations	111
5.5	Attacking Safari with the LVP	115
5.5.1	Value Prediction on Variable Types	115
5.5.2	Speculative Type Confusion	117
5.5.3	End-to-end Attack Overview	122
5.5.4	Evaluation	123
5.6	Attacking Google Chrome with the LVP	125

5.6.1	WebAssembly Function Dispatch Table	125
5.6.2	Speculative Function Confusion	126
5.6.3	Evaluation	129
5.7	Mitigations	131
5.8	Conclusion and Future Work	132
5.9	Ethics Considerations	133
Chapter 6:	Conclusion	134
6.1	Future Work	137
Appendices		140
Appendix A:	LAP Behavior on Page Boundaries	141
Appendix B:	Confirming Instruction Address Tagging	143
Appendix C:	Timer-Resilient Covert Channel	145
Appendix D:	macOS Development Kernel Setup	146
Appendix E:	Extended Tests for Isolation	147
Appendix F:	FLOP-Data Implementation Details	149
Appendix G:	Cache Covert Channel Details	151
Appendix H:	FLOP-Control Implementation Details	152
References		154

LIST OF TABLES

3.1	The L1-I cache organization of various Apple CPUs. W : Ways, S : Sets, CL : Cache Line size.	24
3.2	The L1-D cache organization of various Apple CPUs. W : Ways, S : Sets, CL : Cache Line size.	24
3.3	The L2 cache organization of various Apple CPUs. W : Ways, S : Sets, CL : Cache Line size. –: the pLRU-based minimal eviction set algorithm did not resolve, and we could not confirm using an alternative information source. †: these devices use a random replacement policy for the L1 cache [88], precluding us from using the eviction set algorithm. *: we could not confirm the results, but these are the most likely based on other Apple CPUs with similar cache organization.	25
3.4	Probability of a correct observation (out of 1000 runs) by our distinguisher gadget, across various timer resolutions.	30
3.5	Timerless Spectre-v1 performance across browsers.	34
3.6	End-to-end performance of iLeakage.	45
4.1	LAP presence/absence on recent Apple devices.	66
4.2	Results for Spectre-LAP and ASLR break attacks on the M2 and M3 CPUs, shown as the median of 100 trials. The baseline accuracy, i.e., randomly guessing the ASLR slide, is $1/5120 \approx 0.0002$	81
5.1	Survey of LVP presence on recent Apple desktop and mobile CPUs.	103
C.1	Average runtime and standard deviation (both in milliseconds) for the NOT gate-based cache amplification primitive on native and WebAssembly run-times on the Apple M2 CPU.	145

LIST OF FIGURES

3.1	The access times when accessing the same offset in a loop for a varying number of pages to determine the ways of the L1-D cache set and the L1-I cache set on the P-cores and E-cores of various Apple CPUs. The round markers indicate the detected number of ways.	22
3.2	The access times when accessing the same offset in a loop for a varying number of pages to determine the stride between each L1-D cache set and the L1-I cache set on the P-cores and E-cores of various Apple CPUs. The round markers indicate the detected stride.	23
3.3	The number of <code>mul</code> instructions that can be executed speculatively when the condition is evicted from the L1 cache (dashed) vs. the L2 cache (solid) on various CPUs.	26
3.4	An overview of how our race condition-based gadget runs to distinguish cache hits from cache misses.	28
3.5	Comparison of eviction set-finding algorithms (ours and [25]) across degrading timer resolutions.	32
3.6	L1 cache eviction rate of a victim when evicting its L2 cache line from another core on the Apple M1 CPU.	33
3.7	Memory layout of JavaScript strings in WebKit. The <code>JSCell</code> class is common to all objects, while the <code>StringImpl</code> class is the backing store specific to strings.	37
3.8	NaN-boxing in WebKit <code>JValues</code> to discern data types. The bit patterns that indicate each data type are colored in red, while <code>x</code> marks attacker-controllable bits.	39
3.9	(Top) Pseudocode to construct a malicious object for transiently dereferencing the target address <code>addr</code> . (Bottom) A comparison with the memory layout of a <code>string</code> object.	40

3.10	Ideal placement of the <code>JSCell</code> , with a new cache line between <code>StructureID</code> and <code>Butterfly Ptr</code>	42
3.11	Comparison of memory offsets between a string object and an <code>Intl.Locale</code> object whose <code>JSCell</code> class is split across cache lines, and whose indexed properties mimic the backing store of a string object. We use the latter for speculative type confusion.	42
3.12	(Top) An email displayed in Gmail’s web view. (Bottom) Recovered sender address, subject, and content.	46
3.13	(Top) Text message sent to an Android phone which has been paired to the Google Messages webpage. (Bottom) Recovered text message in highlights.	47
3.14	(Top) Website displaying the target’s IPv4 and IPv6 addresses. (Bottom) Recovered information from the website’s DOM in highlights, including geolocation.	47
3.15	(Top) Google’s accounts page autofilled by LastPass, where the password is <code>googlepassword</code> . (Bottom) Leaked page data with credentials highlighted.	48
4.1	An overview of load address prediction.	60
4.2	The contents of the array in Listing 4.5 for each experiment. The numbered bulletpoints indicate the order of memory accesses, with the first memory access starting at the beginning of the array.	62
4.3	Striding (solid) and Random (dotted) experiment plots for the P- and E-cores of the Apple M1, M2, and M3 CPUs.	63
4.4	Memory contents and access pattern for the SA+RV and Random experiments. The latter is identical to the bottom half of Figure 4.2.	64
4.5	SA+RV (solid) and Random (dotted) experiment plots for the P-cores on the Apple M2 and M3 CPUs.	65
4.6	A graphical overview of the linked list and buffer data structures to cause misprediction by the LAP. These data structures are also used by the code shown in Listing 4.7.	68
4.7	Overview of the outcome of running Listing 4.7 on Figure 4.6. There is a divergence between architectural execution (blue arrows) and speculative execution (red arrows, highlighted area).	69

4.8	Histogram of latencies to the Flush+Reload memory address that corresponds to the secret value.	71
4.9	Heatmap showing the effects of the linked list length and the stride between training load addresses on the likelihood of LAP activation. Values of zero are highlighted in yellow.	72
4.10	Number of LAP activations out of 1,000 runs when extraneous <code>mul</code> instructions are inserted during training.	75
4.11	Number of LAP activations when <code>mul</code> instructions are placed in the speculation window. (Left) The LAP's predicted address is cached. (Right) The predicted address is flushed.	76
4.12	Our modified misprediction primitive that speculatively dereferences double pointers, allowing for a 64-bit out-of-bounds read. The bullhorns indicate that both the dummy and secret data will be transmitted over the covert channel.	78
4.13	Our changes to the lower half of Figure 4.12 for the pointer values in the buffer to point to dummy and secret functions.	80
4.14	Graphical overview of the browser-based version of our LAP-training gadget. Architectural execution is shown in blue arrows, while speculative execution is shown in red arrows and the highlighted region.	83
4.15	Diagram of the regions of WebKit heap memory that are and are not reachable with our attack primitive.	85
4.16	Simplified memory layouts of WebKit's address space. (Top) No memory pressure is applied. (Middle) Memory pressure is applied with the filler strings, and the target page has not loaded yet. (Bottom) Memory pressure is applied, and the target page has completely loaded.	86
4.17	(Top) Email subject and sender name shown as part of Gmail's DOM. (Bottom) Recovered strings from this page.	88
4.18	(Top Left) A listing for coffee pods from Amazon's 'Buy Again' page. (Bottom Left) Recovered item name from Amazon. (Top Right) A comment on a post on Reddit, and (Bottom Right) the recovered text.	88
5.1	Overview of load value prediction.	99

5.2	Comparison of our experiment against its control. While the access order for load addresses is always random, we vary the content of the <code>mem</code> buffer to be constant in our experiment and random in our control.	101
5.3	Runtimes of the gadget in Listing 5.12 on each CPU and core type. The control and experiment plots heavily overlap for the E-cores.	102
5.4	The effect of load width on LVP activation for the M3 CPU.	104
5.5	Runtime when the M3 CPU loads values that stride, instead of the same value on each load. Our control continues to load random values.	105
5.6	Effect of loop unrolling on LVP activation. We attribute the longer latency on the unrolled plot to the much larger binary size, as more instructions must be fetched by the CPU.	105
5.7	Maximum number of distinct load instructions the LVP can support before a timing spike is observed, depending on the instruction distance between them. In the 64-bit Arm ISA, every instruction is 4B wide.	106
5.8	Graphical summary of the code in Listing 5.13. The LVP’s misspeculation and subsequent rollback causes <code>foo</code> to be transmitted, followed by <code>bar</code> (indicated by the satellite icons).	108
5.9	Effect of the number of training loads on the number of observed LVP activations with the stale load value (out of 1000).	109
5.10	Number of observed LVP mispredictions when busy-waiting between training and misprediction, with and without a memory-intensive workload running on the same CPU core.	110
5.11	Number of observed LVP mispredictions when extra <code>mul</code> instructions are inserted between the speculative load and covert channel transmission. The speculation window is much larger when the load target is flushed.	111
5.12	Graphical summary of our modified setup with indirection via an array of pointers to test for out-of-bounds reads during LVP speculation.	112
5.13	Modified lower half of Figure 5.12 to cause the stale load value <code>foo</code> to retrieve a function pointer and branch to it.	114
5.14	Memory layout of the <code>JSCell</code> header.	115

5.15	Memory layout of JavaScript typed arrays in WebKit. The Uint8Array (a class of typed array) declaration at the top of the figure produces this layout with a 3-byte buffer holding the data.	117
5.16	Memory layout of a small JavaScript object with an inline property (Top) and typed array split across cache lines (Bottom).	118
5.17	Memory layout of a small JavaScript object with a string (<code>objWithStr</code>) and typed array containing a malicious payload designed to imitate the string (<code>evil</code>). In this figure, the string's backing store (pointed to by <code>StrData</code>) is not shown for simplicity. We note that the Apple M3 is a little-endian CPU, hence we write the raw bytes in reverse order.	119
5.18	The full memory layout of the JavaScript string "hello", which is the <code>.prop</code> property of the <code>objWithStr</code> object in Figure 5.17 above.	120
5.19	Our completed speculative type confusion primitive.	121
5.20	Sensitive data recovered with FLOP-Data from Google Maps Timeline (Top), Proton Mail's inbox (Middle), and iCloud Calendar (Bottom).	124
5.21	Memory layout of Chrome's function dispatch table.	125
5.22	Our attack routine that causes two LVP mispredictions, disrupting control flow towards an incorrect function and also with the wrong arguments. . . .	127
5.23	(Left) UI elements from Square's customer account page for a storefront. (Right) Recovered last 4 credit card number digits, expiration date, and billing address via FLOP-Control.	130
A.1	Diagram of how the buffer used for training the LAP is allocated across pages for each case, along with the number of activations to the right. The 'D' and 'S' squares represent dummy and secret values, respectively. . . .	141
B.1	(Left) Number of LAP activations when training is interrupted across a function call, starting at the last linked list node and ending at the 30th-to-last. (Right) Number of LAP activations when all but the last 30 nodes are traversed by the original function, and the rest are traversed by the aliased clone function.	144

LIST OF ACRONYMS

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ASID	Address Space Identifier
ASLR	Address Space Layout Randomization
BTB	Branch Target Buffer
COEP	Cross-Origin Embedder Policy
COI	Cross-Origin Isolation
COOP	Cross-Origin Opener Policy
CPU	Central Processing Unit
DIT	Data Independent Timing
DMP	Data-dependent Memory Prefetcher
DOM	Document Object Model
eTLD+1	Extended Top Level Domain, Plus One Prefix
GPU	Graphics Processing Unit
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IBTB	Indirect Branch Target Buffer
IPC	Inter-Process Communication
ISA	Instruction Set Architecture
JIT	Just In Time
KDK	Kernel Debug Kit
L1D	Level 1 Data Cache

L1I Level 1 Instruction Cache
LAP Load Address Predictor
LVP Load Value Predictor
PAC Pointer Authentication Code
PC Program Counter
PID Process Identifier
pLRU Pseudo Least Recently Used
PoC Proof of Concept
PSL Public Suffix List
RAM Random Access Memory
RAW Read After Write
RSB Return Stack Buffer
SA+RV Striding Addresses from Random Values
SMT Simultaneous Multithreading
TSX Transactional Synchronization Extensions
URL Uniform Resource Locator
WAR Write After Read
WAW Write After Write

SUMMARY

Improving the performance of modern processors faces challenges in frequency, power consumption, and relatively slower memory subsystems. In response, architects have devised complex caching, speculation, and prefetching mechanisms for cutting-edge processors to sidestep such limitations. The resulting products have paved the way for an ecosystem of high-performance web applications, leading to users ‘living in the browser’ for several hours each day. In turn, these habits have led to web browsers becoming a central store for users’ secrets, such as passwords and payment information.

This thesis explores the effect of recent hardware optimizations in processors on the security of web applications rendered by browser engines. Building on the knowledge that some hardware optimizations may backfire for isolation across security domains in the form of microarchitectural side-channels, this thesis discovers optimizations that were previously unseen in production, and demonstrates that both old and new optimizations carry consequences that permeate through the layers of computing abstraction – all the way to high-stakes application software on top.

The multifaceted attacks that cause the disclosure of users’ secrets in this thesis pro- pound that the current landscape of retroactive and incremental side-channel mitigations in web browsers are insufficient to defend against future threats, especially by demonstrat- ing how to undermine several such countermeasures with novel hardware and software primitives. By characterizing the adversarial capabilities of web-based microarchitectural threat actors, this thesis intends to serve as an initial step towards principled defenses for hardening future web browsers.

CHAPTER 1

INTRODUCTION

Since the 1970s, Moore's law has anticipated exponential growth in microprocessor performance from rapidly shrinking process nodes, and the Central Processing Unit (CPU) industry has empirically followed suit. However, Moore's law faces two challenges in the present day. The first is that process nodes are approaching limits defined by physics for how narrow a circuit electrons can pass through, constraining the growth in transistors per chip area. To achieve performance growth despite this, CPU architects are devising new and more complex prediction and prefetching mechanisms in microarchitecture. Moreover, the second challenge is that the performance of memory systems has not grown at the same rate, but rather significantly slower. In response, recent CPUs have grown in cache size to better hide the memory access latency, and developed deep speculative and out-of-order execution mechanisms to avoid pipeline stalls.

On the other hand, this rapid improvement in CPU performance has also coincided with the birth and explosive growth of the Internet since the 1990s. This has fueled an extensive migration of native applications to within the web browser (e.g., email clients and office suites) and an emergence of new web applications (e.g., online banking and social media), leading to a flourishing ecosystem. However, a major consequence of people 'living in the browser' on a daily basis is that the browser has become a central bastion of secrets, such as passwords, payment information, and contact information. Hence, in the present day, securing browsers is an utterly important task. In turn, modern browsers employ several security measures to protect these secrets from attacker-controlled webpages, such as using process isolation for mutually distrusting origins and executing JavaScript and WebAssembly code in software sandboxes.

Despite these isolation mechanisms, secrets stored in web browsers are unfortunately

susceptible to side-channel attacks. Due to mutually distrusting domains executing on the same underlying hardware, the implicit sharing of hardware resources allows one domain to infer activity on the other domain by monitoring variables such as execution time or resource contention. For modern CPUs in particular, the risk of microarchitectural side-channels such as Spectre [1] and Meltdown [2] becomes exacerbated due to them sharing multiple cache, predictor, and prefetcher states across security domains, as well as speculating on control and data dependencies, all in the pursuit of performance.

Accordingly, this thesis contributes to understanding when and how these performance optimizations backfire with security implications, resulting in attack vectors. These attack vectors are constantly emerging, but are unknown to browser developers because microarchitecture is largely invisible to software and proprietary to the CPU vendor. Therefore, amidst the many layers of abstraction between hardware and browser environments, the objective of the works in this thesis is to first characterize the interactions between the two layers, and then apply this knowledge to harden the next generation of CPU hardware and browser software against the adversarial capabilities which they demonstrate.

1.1 Thesis Organization and Contribution Summary

Firstly, **chapter 2** is comprised of related work and high-level concepts that are common to all works in this thesis, as to provide context for subsequent chapters.

In **chapter 3**, this thesis studies the resilience of the Apple ecosystem to speculative execution attacks, 4-5 years after the disclosure of Spectre [1]. We find that Apple’s closed and homogeneous ecosystem results in side-channel attack surfaces of unprecedented magnitude, where the use of the same Arm CPU cores from lightweight iPhone/iPad to heavy-weight Mac applications (with the quantity of cores determining the device’s compute performance) facilitates migrating our iLeakage attack between desktop and mobile platforms, and where App Store policy that mandates the use of the WebKit browser engine for all iOS web browser apps causes trivial migrations to browsers outside of Apple Safari (such

as Chrome and Firefox). This is in sharp contrast to prior work being bifurcated between x86-based desktop platforms and Arm-based mobile platforms, and prior exploits being engineered for one targeted application.

The iLeakage attack further demonstrates that two browser countermeasures to Spectre are insufficient by recovering the target’s Gmail inbox, Android text messages, and Google account credentials as a remote web-based adversary who owns an arbitrary domain. While Apple, in an attempt to thwart cache attacks, had restricted the available timer resolution in WebKit to be seven orders of magnitude coarser than the difference between cache hit and miss (with other browser vendors implementing similar measures), we show that race conditions in hardware can serve as a cache oracle, allowing for timerless eviction set generation and Spectre Proof of Concept (PoC)s. Moreover, though WebKit renders each tab in a separate process such that Spectre-v1 cannot read cross-origin data, we show that a benign web Application Programming Interface (API) feature for popups and federated logins can be abused to force the attacker and arbitrary target’s webpages into one process.

In **chapter 4**, we move away from undermining countermeasures for existing microarchitectural attack surfaces, and towards understanding how novel performance optimizations can open unforeseen attack surfaces. We uncover that a Load Address Predictor (LAP) has been implemented in a production CPU for the first time. The LAP monitors strides in the address of previous memory loads, and then speculatively performs the next load on a predicted address. During this speculation window, the CPU can execute arbitrary instructions on the data from the predicted address. Hence, the LAP manifests as a medium for speculative execution on data hazards, where the real-world presence of such mediums has been rare. This augments Spectre’s insight, and that of many follow-up works, that CPUs can execute the wrong control flow and leak secrets when attempting to resolve control hazards such as conditional branches, indirect branches, and returns.

We demonstrate that, as an analogy, CPUs can execute the wrong data flow when the LAP is mistrained, and perform arbitrary computations on data-at-rest that should never

have been brought into the CPU core. Although Apple’s LAP implementation had limited the stride (and thus out-of-bounds reach) to 255 bytes, we show that pointer-based indirection can be used in native code to read the entire address space and execute rogue functions using code gadgets that merely look like linked list traversals (as opposed to a Spectre gadget). Moreover, we show the LAP’s real-world risks for web browser code. Augmenting our findings from chapter 3, we discover that not only can arbitrary webpages share a process in Safari, but they can also share memory heaps for strings within the LAP’s stride limit. This allows an attacker webpage to recover sensitive cross-origin strings even when architecturally reading just their own data, and despite pointers not being available in JavaScript for indirection techniques.

We dive deeper into the perils of data speculation in **chapter 5**, where we uncover a mechanism in the wild that speculates past Read After Write (RAW) dependencies, the fundamental data dependency that even out-of-order execution and register renaming cannot resolve. This chapter investigates the first real-world implementation of a Load Value Predictor (LVP), which monitors data values returned from the memory subsystem after loads as opposed to memory addresses. If the next load cannot be resolved quickly, the LVP injects a predicted data value into the CPU pipeline and executes younger dependent instructions using the prediction until the ground-truth data arrives from main memory. We observe that this behavior causes computations using rogue values under speculation, but not for 64-bit loads, presumably out of concern that they could be pointers.

Nonetheless, we demonstrate that rogue 32-bit load data are still dangerous to web browser security, especially when they are used to encode security-relevant information such as function signature and object type. That is, we cause the LVP to mispredict the signature value for WebAssembly functions in Google Chrome, resulting in the function being provided incorrect arguments. Moreover, we show that Apple’s Safari mitigations following our disclosure of the work in chapter 3 are undermined by the LVP’s presence, as it provides an additional method to (speculatively) circumvent the type checks for JavaScript

objects. This results in arbitrary 64-bit read capabilities for adversaries in both browsers, resulting in the disclosure of location history, email subjects, billing addresses, and more.

Finally, **chapter 6** summarizes how future hardware and web browsers could defend against the adversarial techniques presented in this thesis, and lays out potential directions for future work.

CHAPTER 2

BACKGROUND

First, we discuss the current state of research in two fields that are common to the works in this thesis, as to supplement and contextualize the background section of each work.

2.1 Sources of Speculative Execution

With the everlasting pursuit for compute performance by the processor industry and the increasing performance gap between processors and memory, computer architects have diversified the microarchitectural components for which speculative execution can be applied, and thus for which potentially useful computations can be performed while waiting for the memory subsystem. While the original disclosure of Spectre [1] uncovered the security consequences of speculation on conditional branches (also by [3, 4, 5, 6]) and indirect branches, a plethora of subsequent work has complemented the research area which Spectre shaped.

Indirect branches have been of particular focus recently for two reasons: that they can be exploited across hardware-backed security domains such as processes [7], the user-kernel boundary [8, 9], and virtual machines [10], and that a cat-and-mouse game is ongoing between vendor mitigations and researchers finding new attack techniques to circumvent them [7, 8, 9, 10, 11, 12, 13]. Moreover, speculative attacks on return instructions have been demonstrated [14, 15], with some works finding that they can interact with indirect branch predictions [7, 16].

While speculation on control flow is predominant among attack works, there is also a small body of work uncovering speculation on data flow. Modern processors also speculate on memory disambiguation, that is, performing store-to-load forwarding before confirming that the addresses match [17, 18]. Furthermore, some of them assume that floating point

operands are not subnormal and handle corner cases retroactively in microcode, leading to speculation windows [19, 20]. Lastly, data-dependent prefetching can also be thought of as a form of data speculation, as the prefetch happens only when the CPU anticipates that the data is an address which will be accessed later [21, 22, 23].

2.2 Side-Channel Attacks on Web Browsers

As web browsers have become a centralized point for users' secrets through login-protected web services, they also have become a popular target for exploitation, both by traditional attacks (e.g., buffer overflows and use-after-frees) and side-channel attacks. Here, we focus on the latter. At the basics, cache attacks have been migrated to operate in browser-based code to fingerprint websites when they are rendered in other browser tabs [24, 25, 26, 27, 28], given that the browser will execute untrusted JavaScript or WebAssembly code from an attacker's webpage automatically.

Building on some of these cache primitives for a covert channel, researchers have used speculative execution to bypass the browser's sandboxing measures that are supposed to execute untrusted code safely, achieving arbitrary read capabilities [19, 29, 30, 31]. These reads are often to cross-origin data, violating the same-origin policy which mandates browser-based code to have access only to data served from the same website.

However, hardening the sandbox is still not sufficient to isolate cross-origin data, since the data can be inferred indirectly when it is rendered as pixels on the screen. Websites can load cross-origin content in an `iframe` element, but their scripts cannot access data within the `iframe`. Nonetheless, researchers have overcome this countermeasure with SVG filters which operate on the pixels of the `iframe`, inferring each pixel value in a manner similar to optical character recognition. Such pixel-stealing attacks leverage a variety of side channels for inference, ranging across timing, frequency scaling, hardware compression, and cache activity [20, 32, 33, 34, 35].

Finally, several works show that sandboxing is also insufficient to prevent the migration

of more advanced native side-channel attacks (which requires the attacker to own an unprivileged account on the target machine) to the browser, where the only assumption is that the target visits an attacker-controlled webpage. This includes execution port contention in CPU cores [36], flipping bits in memory with the Rowhammer phenomenon [37, 38, 39, 40], and profiling the execution of cryptographic operations [41].

CHAPTER 3

ILEAKAGE: BROWSER-BASED TIMERLESS SPECULATIVE EXECUTION ATTACKS ON APPLE DEVICES

Over the past few years, the high-end CPU market has been undergoing a transformational change. Moving away from using x86 as the sole architecture for high performance devices, we have witnessed the introduction of computing devices with heavyweight Arm CPUs. Among these, perhaps the most influential was the introduction of Apple’s M-series architecture, aimed at completely replacing Intel CPUs in the Apple ecosystem. However, while significant effort has been invested analyzing x86 CPUs, the Apple ecosystem remains largely unexplored.

In this chapter, we set out to investigate the resilience of the Apple ecosystem to speculative side-channel attacks. We first establish the basic toolkit needed for mounting side-channel attacks, such as the structure of caches and CPU speculation depth. We then tackle Apple’s degradation of the timer resolution in both native and browser-based code. Remarkably, we show that distinguishing cache misses from cache hits can be done without time measurements, replacing timing based primitives with timerless and architecture-agnostic counterparts based on race conditions. Finally, we use our distinguishing primitive to construct eviction sets and mount Spectre attacks, all while avoiding the use of timers.

We then evaluate Safari’s side-channel resilience. We bypass the compressed 35-bit addressing and the value poisoning countermeasures, creating a primitive that can speculatively read and leak any 64-bit address within Safari’s rendering process. Combining this with a new method for consolidating websites from different domains into the same render process, we demonstrate end-to-end attacks leaking sensitive information, such as passwords, inbox content, and locations from popular services such as Google.

3.1 Introduction

The ubiquity of computing devices has resulted in the ever-increasing popularity of web browsers. Indeed, users now consume and create content directly through the browser, using it as a main access point to cloud-backed services and infrastructure. This usage pattern makes the browser one of the most important components of a user-facing computer system, with the browser’s address space routinely containing sensitive information such as login credentials, emails, documents, pictures, etc.

With browsers routinely loading and executing JavaScript code from untrusted sources, many side-channel attacks have used the browser as a convenient starting point for compromising the system through its underlying hardware [1, 14, 19, 20, 24, 27, 28, 29, 30, 31, 37, 38, 42, 43, 44, 45, 46]. As most attacks rely on the ability to measure time of microarchitectural events, vendors in turn have attempted to harden browsers against side channels by severely degrading available timer resolution [47, 48, 49], as well as limiting the use of `SharedArrayBuffers` in some cases [50] to prevent attackers from crafting a high-resolution timer.

Next, the ever-changing computing landscape has recently resulted in the introduction of high-end Apple silicon devices, aiming to compete with their x86 counterparts. While there is a plethora of works analyzing browser-based side channels on Intel and AMD architectures, the Apple ecosystem remains poorly understood despite its popularity [21, 28, 29, 51, 52]. With the growing popularity of Apple’s M-series, we investigate the following main questions:

Is the ability to measure time truly critical for mounting microarchitectural side-channel attacks? In particular, how can attackers mount transient execution attacks inside the browser, potentially without relying on any timing primitives? Finally, how resilient are Apple devices to side-channels? And how can adversaries exploit them?

3.1.1 Our Contributions

In this chapter, we present iLeakage, a speculative type-confusion attack that can extract information from Apple’s Safari browser. In particular, we can defeat Apple’s low-resolution timer, compressed 35-bit addressing, and value poisoning countermeasures, allowing us to read any 64-bit address within the address space of Safari’s rendering process. Combining this with a new technique for consolidating websites from different domains into the same renderer process, we craft an end-to-end attack capable of extracting sensitive information (e.g., passwords, inbox content, locations, etc.) from popular services such as Google. Finally, we note that Safari / WebKit is the only browser engine permitted on iOS devices regardless of web browser app. This makes nearly all smartphone and tablet devices made by Apple susceptible to our attack.

Next, as little is known about transient execution attacks on Apple devices, before constructing our attack we must first develop common (and often timerless) side-channel primitives on Apple silicon. Given the popularity of Apple hardware, and the timerless nature of our constructions, these might be of independent interest.

Investigating Cache Layout. We begin by investigating the topology of the cache hierarchy on Apple CPUs. As this information is not available via unprivileged performance counters, we design a set of experiments for recovering the total size, associativity, cache line size, and inclusiveness of Apple’s Level 1 Instruction Cache (L1I), Level 1 Data Cache (L1D), and L2 caches.

Measuring Speculation Window. We then proceed to measure the number of instructions Apple CPUs can execute under speculation, before the CPU discovers a branch misprediction. Here we show that modern Apple hardware has deep pipelines, with speculation windows running as long as 300 cycles.

Distinguishing Cache Misses from Cache Hits. Our next task is to distinguish cache hits from misses. While this can be achieved typically by measuring time, Apple has limited the clock resolution in both native and browser-based environments. Noting that cache

hits and misses result in different sizes of speculation windows, we present a methodology for replacing timing based primitives with timerless counterparts based on race conditions. Using this approach, we are able to reliably distinguish hits from misses using the native 42 ns timer, Safari’s 1 ms timer, Tor’s 100 ms timer, and even without the use of timers altogether. To the best of our knowledge, this is the first browser-based demonstration of timerlessly distinguishing cache misses from cache hits, and the first primitive that does not rely on Instruction Set Architecture (ISA)-specific features.

Constructing Eviction Sets. We then shift to constructing cache eviction sets. Here, we improve prior work by Vila et al. [25], presenting a new group testing and backtracking approach which allows the algorithm to converge within minutes even without the use of timers. As degrading the timer resolution is often seen as a countermeasure to eviction set construction, our timerless technique might be of independent interest.

A Timerless Browser-based Speculative Attack. We combine all of our above-constructed primitives into a timerless Spectre v1 gadget PoC. At a high level, we achieve this by replacing the cache timing-based method of leaking secrets under speculation with our gadget for timerlessly distinguishing cache misses from cache hits. Here, we show that our attacks have near perfect accuracy, across Safari, Firefox and Tor.

Mounting Transient Execution Attacks in Safari. Next, we proceed to mount speculative side-channel attacks on the Safari browser. We begin by abusing Safari’s site isolation policy, demonstrating a new technique that allows the attacker page to share the address space with arbitrary victim pages, simply by opening them using the JavaScript `window.open` API. We then construct in-browser eviction sets, side stepping Safari’s timer mitigations. Finally, we bypass Apple’s compressed 35-bit addressing and value poisoning countermeasures using speculative type confusion, allowing the attacker to craft and dereference arbitrary 64-bit pointers. Here, we show that assumptions used when designing architectural memory safety approaches do not hold true under speculation.

Leaking Sensitive Data. As a final contribution, we demonstrate the security impli-

cations of the techniques we developed and present end-to-end use cases of our attacks. More specifically, we show how an attacker webpage can open the target page and subsequently read information from it. Empirically demonstrating this, we show recovery of inbox contents and text messages. Next, applying this technique to password managers such as LastPass, we exploit the auto-fill feature to leak the target’s Google credentials. We note that this attack is practical: it depends only on the target visiting the attacker’s website, and runs to completion on Macs, iPhones, and iPads in their default configurations.

Summary of Contributions. We contribute the following:

- We study the cache topology, inclusiveness, and speculation window size on Apple CPUs (subsection 3.4.1, subsection 3.4.2, and subsection 3.4.5).
- We present a new speculative-execution technique to timerlessly distinguish cache hits from misses (subsection 3.4.3).
- We tackle the problem of constructing eviction sets in the case of low resolution or even non-existent timers, adapting prior approaches to work in this setting (subsection 3.4.4).
- We demonstrate timerless Spectre attack PoCs with near perfect accuracy, across Safari, Firefox and Tor (subsection 3.4.6).
- We mount transient-execution attacks in Safari, showing how we can read from arbitrary 64-bit addresses despite Apple’s address space separation, low-resolution timer, caged objects with 35-bit addressing, and value poisoning countermeasures (section 3.5).
- We demonstrate an end-to-end evaluation of our attack, showing how attackers can recover sensitive website content as well as the target’s login credentials (section 3.6).

3.1.2 Ethics and Artifact Availability

We initially disclosed our findings to Apple’s product security team on September 12, 2022. Apple has acknowledged the issues, requesting an embargo on the paper’s contents. We have actively discussed countermeasures with Safari’s security team and maintained contact with Apple’s head of product security. Our discussion has resulted in Apple refactoring Safari’s multi-process architecture significantly, which we detail in section 3.7. At the time of writing these changes are under active development, and are available in Safari Technology Preview versions 173 and newer.

3.2 Background

3.2.1 Caches and Cache Attacks

To bridge the increasing performance gap between the CPU and main memory, the CPU contains small buffers called *caches*. These exploit locality by storing frequently and recently used data to hide the memory access latency. We primarily focus on recent Apple Silicon CPUs, which feature a heterogeneous core design with one or more clusters of high-performance P-cores and energy-efficient E-cores. While each core type differs in its cache organization, they have a private L1 cache and a shared L2 cache per core cluster.

Cache Associativity. Typically, these caches are divided into multiple cache sets that can host up to a certain number of cache lines or *ways* or *associativity*. Part of the virtual or physical address is then used to map a cache line to its respective cache set, where *congruent* addresses are those that map to the same cache set.

Cache Attacks. By monitoring the target’s cache accesses, an attacker can infer secret information from the target in a shared physical system. Previous works proposed many different techniques to perform such attacks, most notably Flush+Reload [53, 54, 55] and Prime+Probe [24, 25, 56, 57, 58, 59, 60, 61, 62]. Most aforementioned cache attacks on Intel CPUs have targeted the last-level (L3) cache, as it is shared among all cores and an L3

cache miss incurs measurable spikes in latency. We instead target the L2 cache on Apple CPUs, but for the same rationale.

3.2.2 Speculative and Out-of-Order Execution

Rather than following the strict program order, modern processors execute instructions as soon as the required data is available, a concept called out-of-order execution. Furthermore, to handle branches whose condition is yet to be resolved, the processor attempts to predict the branch condition, speculatively executing instructions along the corresponding path. When the condition is eventually computed, if the branch has been mispredicted, the processor will revert the speculatively executed computation and will proceed to execute the correct path instead.

The discovery of Spectre [1] and Meltdown [2] showed that speculative execution has security implications. Specifically, while the processor can reverse all architectural effects resulting from incorrect speculation, microarchitectural effects such as cache and predictor states are not restored. Transient-execution attacks leverage this partial state reversal to extract information not available otherwise to the attacker, violating the separation between many mutually-distrusting hardware-backed security domains [2, 3, 5, 6, 11, 14, 15, 16, 17, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76].

Finally, in concurrent work, Katzman et al. [77] show that speculative execution can even be used to build logic gates that operate on the cache state, which in turn can amplify cache attacks.

3.2.3 Side Channel Attacks on Apple CPUs

While CPUs made by Intel and AMD have received a generous amount of side channel research attention, much less is known about side channel vulnerabilities in Apple's Arm-based A- and M-series CPUs. More specifically, Shusterman et al. [28] demonstrate cache capacity attacks on the Apple M1 while Leaky.page [29] and Hetterich et al. [78]

established the feasibility of Spectre v1 exploits on these machines. More recently, the m1racles [51] attack demonstrated a cross-process covert channel due to insufficient access control to system registers, while Hot Pixels [33] showed that data-dependent frequency scaling on Graphics Processing Unit (GPU)s and CPUs enables pixel stealing and website fingerprinting attacks.

Next, Augury [21] demonstrated the existence of a Data-dependent Memory Prefetcher (DMP) on the Apple M1, allowing attackers to bypass some Spectre countermeasures as well as derandomize the kernel’s Address Space Layout Randomization (ASLR). Finally, PACMAN [52] shows how attackers can forge kernel Pointer Authentication Code (PAC)s from userspace to bypass pointer authentication on M1 CPUs.

3.2.4 Side Channel Attacks in Web Browsers

Side-channel attacks have also been demonstrated using browser-based code (e.g., JavaScript and WebAssembly). Indeed, cache attacks running within browser tabs have been used for cryptographic key extraction [42], monitoring user activity [24, 27, 28], and even to cause Rowhammer [37, 38, 43]. Next, the data dependency of floating point instructions was also exploited in the browser, in the form of pixel stealing attacks [20, 44, 45]. Finally, early transient execution vulnerabilities have also been demonstrated in browsers [1, 14, 46], prompting large mitigation efforts.

Transient Execution Attacks in Chrome. Following the demonstration of the original Spectre [1] attack in JavaScript, Google attempted to harden Chrome against transient execution attacks. This effort led to the deployment of site isolation, where different websites are rendered in different address spaces unless they are subdomains of the same parent site.

Leaky.page [29] shows that this hardening is necessary, as attackers can reliably mount Spectre-based attacks on modern versions of Chrome, albeit being limited to reading specific 4 GB heaps in Chrome’s 64-bit address space. Finally, Spook.js [31] overcomes this 4 GB restriction, allowing attackers to read browser secrets assuming the attacker and vic-

tim pages share the same Extended Top Level Domain, Plus One Prefix (eTLD+1) domains.

In this chapter, we show that disclosure of browser secrets is still possible and practical even with the degradation or removal of timing sources, through the timerless nature of our primitives. More specifically, we eliminate the dependency on `SharedArrayBuffer`-based timing primitives of Spook.js, shown to have approximately 5 ns resolution [26]. Remarkably, timers with 5 ns resolution are unavailable even to native code in Apple CPUs, let alone JavaScript in browsers: we detail this landscape in subsection 3.2.5.

Transient Execution Attacks in Firefox. Transient-execution attacks have also been demonstrated within modern versions of the Firefox web browser. Ragab et al. [19] showed how incorrect results generated transiently by floating point units in Intel and AMD machines can lead to type-confusion attacks, allowing the attacker to dereference arbitrary 64-bit pointers. More recently, the Spring attack [30] exploits mispredictions in the Return Stack Buffer (RSB) to again demonstrate type-confusion attacks on Firefox.

3.2.5 Timing Sources on Apple CPUs

A common requirement for mounting microarchitectural attacks is the capability of measuring the execution time of different instructions. We now summarize the state of time measuring capabilities in the Apple and web browser ecosystems.

Timers in Native Environments. Both macOS and iOS allow privileged native code to read a cycle-accurate timer which is accessible via the `kperf` API [79, 80]. However, unprivileged code can only access the timer provided by `mach_absolute_time()`, which PACMAN reported to have a resolution of 24 MHz [52], or 42 ns. Such a resolution is intractable for microarchitectural side channels, as most instructions executing in a single-digit number of cycles while cache accesses completing in the dozens.

Timers in Web Browsers. With timers being a critical component for side channel attacks, web browsers typically further restrict the timer resolution available to browser-based JavaScript code [47, 48, 49]. Here, Google Chrome provides a timer resolution of

100 μ s, while Mozilla’s Firefox and Apple’s Safari provide a 1 ms timer. In an effort to resist fingerprinting of any sort, Tor Browser further coarsens the timer to 100 ms.

In older browser versions, one could craft a timer with 5 ns resolution [26] using the `SharedArrayBuffer` API in JavaScript, in a manner similar to a counting thread. However, following the use of high-resolution timers in cache covert channels as part of the Spectre and Meltdown attacks publicized in January 2018, all major browser vendors disabled `SharedArrayBuffer` [50].

Yet, `SharedArrayBuffer` was conditionally re-enabled in all major browsers by December 2021 for webpages served with cross-origin isolation headers [50], as these aim to mitigate Spectre by preventing webpages from loading cross-origin content not explicitly allowed by the server [81]. See subsection 3.5.1 for implications of this condition on the Safari browser.

3.2.6 Timer-Friendly Covert Channels

Transient-execution attacks typically leak values obtained during incorrect speculation via microarchitectural covert channels. However, as browser vendors heavily restrict the timer resolution available to JavaScript code [47, 48, 49], we require a “timer-friendly” covert channel, which can reliably transmit values from the speculative domain even in the case of degraded timers.

pLRU-based Channel. Early works have resorted to timing techniques based on the clock edge [26, 45, 82] to overcome this challenge. More recent work by Google [29] abuses the Pseudo Least Recently Used (pLRU) eviction strategy of L1-D caches to construct a particularly stable covert channel on Intel and Apple CPUs.

For the remainder of this chapter, we abstract the pLRU covert channel as providing four primitives. `plru.init()` initializes the covert channel, and `plru.transmit()` transmits a bit via some microarchitectural state. `plru.traverse()` traverses an L1-D cache set in a way that this routine will take a longer time to execute if `plru.transmit()`

Listing 3.1: Details of the `plru.receive` function.

```
1 function plru.receive() {  
2     runtime = badtimer.time(plru.traverse());  
3     return runtime > threshold ? 1 : 0;  
4 }
```

had previously transmitted a 1 bit, than if a 0 bit was transmitted or no transmission happened at all. Notably, this timing difference can be amplified significantly to be observable with a low-resolution timer by performing sufficiently many traversals over the L1-D cache set, thereby allowing us to use `plru.traverse()` to recover the transmitted bit.

Indeed, Listing 3.1 outlines the recovery procedure, which we call `plru.receive()`. It first times `plru.traverse` with the low-resolution timer (Line 2). Then, leveraging the amplification property, the transmitted value is recovered by comparing `plru.traverse`'s runtime against a coarse-grained calibrated threshold in Line 3.

Amplification with pLRU. Next, the pLRU strategy can be used to amplify the outcome of certain race conditions [83], resulting in the ability to construct eviction sets in the presence of $5 \mu\text{s}$ timers on x86 CPUs. Finally, in concurrent and independent work, Purnal et al. [84] show an improved `plru.traverse` routine over an L2 cache set whose elements are congruent to the L1-D cache set on Intel CPUs, increasing the maximum amplification of the pLRU-based channel from $500 \mu\text{s}$ to 5 ms.

Timerless Channels. Disselkoen et al. [60] use Intel's Transactional Synchronization Extensions (TSX) to mount L3 cache attacks without a timer. However, TSX has been disabled since 2021, owing to the discovery of several side-channel security issues [85]. Next, Zhang et al. [86] observed that some of the newest Intel processors contain monitoring instructions such as `umwait` that are exploitable to build a timerless covert channel. While [86] reported a similarly-behaved instruction on an Arm Cortex-A73 CPU, it is unknown if Apple CPUs are susceptible. Moreover, the `umwait` instructions are not available to

browser-based code.

Stepping away from mounting cache attack, Chen et al. [87] attempts to protect SGX applications from Simultaneous Multithreading (SMT)-level side channels by ensuring that both threads are occupied by the applications' workload. As core scheduling is not exposed to enclaves, [87] achieves this by inducing race conditions on L1-cached variables arising from cache coherence, exploiting the fact that the L1 cache is shared among two sibling threads on Intel machines.

3.3 Threat Model

In this chapter we focus mainly on Apple hardware. For the experiments presented in section 3.5 and section 3.6, we assume that the target has been fully updated with Apple's MacOS 13.1 and iOS 16.2 (latest at the time of writing). In particular, we assume that side-channel countermeasures are left in their default enabled state and that the machine has no (known) software vulnerabilities.

Next, for the attacks presented in section 3.6, we assume a typical model for web-based attacks, where the target visits an attacker-controlled website using the Safari web browser. Here we note that while macOS-based devices allow the installation of other browsers (e.g., Chrome), Apple prohibits non-Safari based browsers on iOS devices. In particular, all browsers installed on iOS devices must use WebKit as their underlying rendering engine, making nearly all modern iOS devices vulnerable to our attack.

3.4 Microarchitectural Primitives

We begin by exploring the status of basic primitives required for mounting micro-architectural attacks on Apple devices:

[P_1] Identifying Cache Organization. We first need to determine the cache organization on Apple CPUs, as our microarchitectural primitives rely on some of these properties.

Thus, we present a collection of primitives to help uncover the cache organization of Apple’s Arm-based CPUs without any elevated privileges.

[P_2] Measuring Speculation Depth. Another building block for our microarchitectural primitives is speculation. Knowing the cache layout allows us to evict data from each cache level, and subsequently measure the number of instructions that can execute speculatively before the data becomes reinstated.

[P_3] Distinguishing Cache Hits From Misses Without Timers. Tackling the degradation of timer resolution as a widespread side-channel countermeasure, we present a primitive that uses speculation to distinguish cache misses from hits at each level of cache hierarchy using only low-resolution timers. We then show an improved variant of this primitive that does not use timers at all, instead using race conditions and shared memory to distinguish between cache hit vs. miss latencies.

[P_4] Constructing Minimal Eviction Sets. We use our cache distinguishing primitive to develop a method for reliably finding minimal eviction sets amid the degradation or absence of timers. We improve on the group-testing eviction set finding algorithm [25] in ways that add resiliency, allowing us to integrate with [P_3].

[P_5] Testing for L2 Inclusiveness. We combine [P_3] and [P_4] to determine whether the L2 cache is inclusive of the L1 caches.

[P_6] Timerless Spectre Attacks. Finally, we demonstrate a Spectre-v1 gadget that can reliably leak data amid severely degraded or completely lacking timers.

3.4.1 [P_1]: Cache Organization

We now proceed to reverse engineer the cache topology of Arm-based Apple CPUs. While kernel extensions can read model-specific registers to recover this information, doing so on sandboxed iOS devices is rather challenging. Thus, we now describe our empirical method to determine the cache topology.

Determining the Associativity of the L1 Cache. Since L1 caches are typically virtually

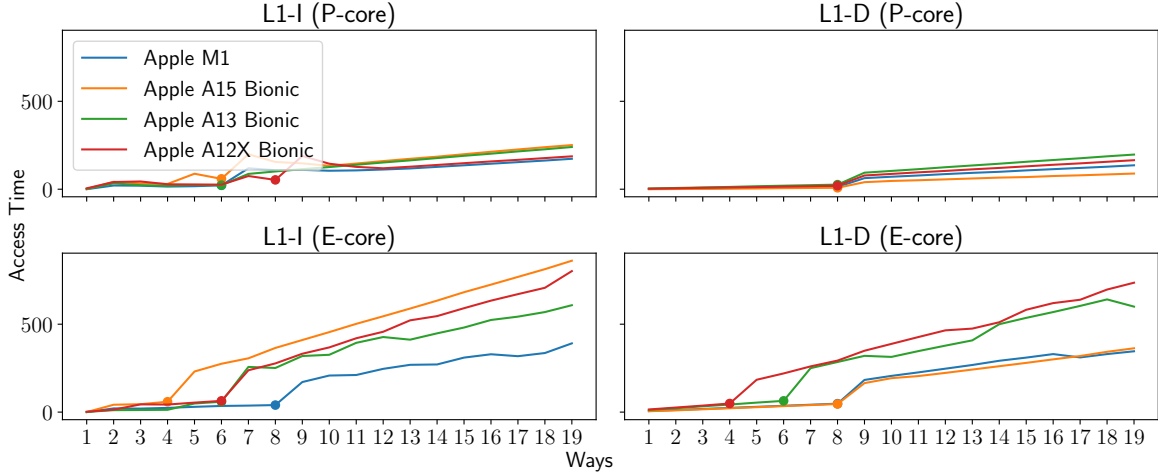


Figure 3.1: The access times when accessing the same offset in a loop for a varying number of pages to determine the ways of the L1-D cache set and the L1-I cache set on the P-cores and E-cores of various Apple CPUs. The round markers indicate the detected number of ways.

indexed, identical page offsets of multiple pages are *congruent*: that is, they map to the same cache set. To determine the associativity a , we measure the runtime to loop through an increasing number of congruent addresses n , where a slowdown will occur if $n > a$. For the L1 instruction cache, we achieve this using chains of branch instructions which are all at congruent addresses. Figure 3.1 summarizes our findings, showing the associativity of the L1 caches on various Apple CPUs.

L1 Cache Size. To parallelize address translation with cache lookups, L1 caches typically limit the indexing bits to the page offset bits. With Apple CPUs using 16KiB pages, we can upper-bound the size of the L1 cache, s , as $s \leq 16 \text{ KiB} \cdot a$. Next, cache designs often try to avoid self-eviction of blocks of continuous memory. Thus, we first allocate a block of size $16 \text{ KiB} \cdot a$, and repeatedly access $a + 1$ elements from it using a given stride δ . We keep doubling δ until we observe a slowdown, indicating the presence of self-evictions. Figure 3.2 presents the access time across different strides. As the largest stride with fast access times, δ^* , still allows $a + 1$ elements to map to different sets, each cache way has δ^* capacity. Thus, the size of the entire cache is $s = a \cdot \delta^*$.

L1 Cache Line Size. To determine the cache line size, we loop through n' congruent

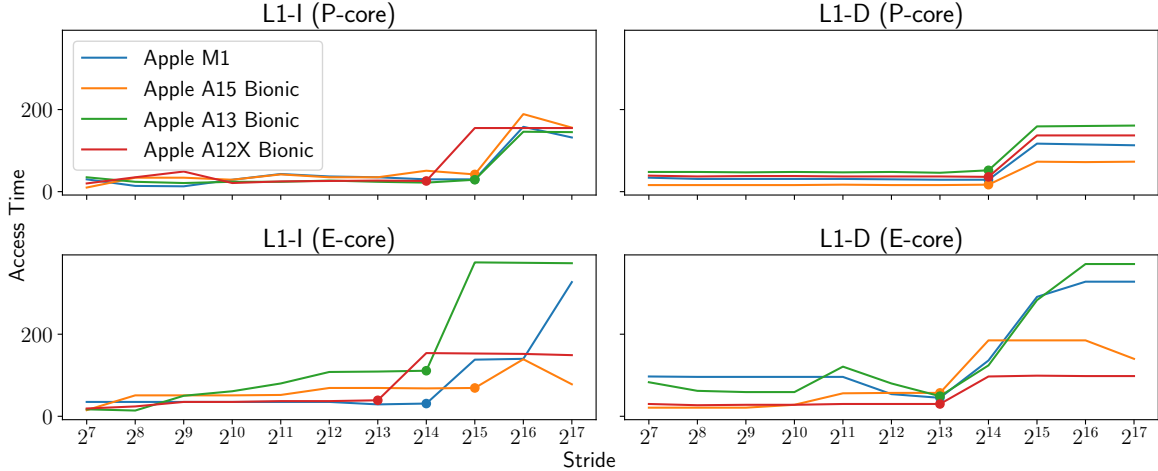


Figure 3.2: The access times when accessing the same offset in a loop for a varying number of pages to determine the stride between each L1-D cache set and the L1-I cache set on the P-cores and E-cores of various Apple CPUs. The round markers indicate the detected stride.

addresses, where $n' > a$ and causes a slowdown. Then, we shift $\frac{n'}{2}$ addresses by an increasing offset. The smallest value of this offset where we observe a relative speedup is the cache line size l , since it will result in $\frac{n'}{2}$ addresses mapping to the next line, and thus the next cache set. Finally, once l is known, we can compute the number of sets t , since $t = \frac{s}{la}$.

L2 Cache Organization. The `sysctl` interface on macOS and iOS provides the line size l and total size s of the L2 cache. However, as the L2 cache is physically indexed and we therefore cannot retrieve a or t here, we rely on the ability to find minimal eviction sets, as its size must equal a . We detail this in subsection 3.4.4.

Experimental Results. Running the above methodology across multiple iOS and MacOS devices, we were able to recover and document the organization of the L1 caches. Table 3.1, Table 3.2, and Table 3.3 present a summary of our findings across multiple generations of Apple CPUs, both for P-cores and E-cores.

3.4.2 $[P_2]$: Measuring Speculation Depth

Knowing the cache organization on modern Apple platforms, we now measure the number of instructions that can execute speculatively, also known as the speculation window length.

Table 3.1: The L1-I cache organization of various Apple CPUs. **W**: Ways, **S**: Sets, **CL**: Cache Line size.

Name	CPU	L1-I Cache			
		W	S	Size	CL
MacBook Air (M2, 2022)	M2 (P)	6	512	192 KiB	64B
(Mac14,2) A2681	M2 (E)	4	512	128 KiB	64B
MacBook Pro 14"	M1 Max (P)	6	512	192 KiB	64B
(MacBookPro18,4) A2442	M1 Max (E)	8	256	128 KiB	64B
MacBook Pro 14"	M1 Pro (P)	6	512	192 KiB	64B
(MacBookPro18,3) A2442	M1 Pro (E)	8	256	128 KiB	64B
Mac Mini (M1, 2020)	M1 (P)	6	512	192 KiB	64B
(Macmini9,1) A2348	M1 (E)	8	256	128 KiB	64B
iPhone 13 mini	A15 Bionic (P)	6	512	192 KiB	64B
(iPhone14,4) A2481	A15 Bionic (E)	4	512	128 KiB	64B
iPhone 12	A14 Bionic (P)	6	512	192 KiB	64B
(iPhone13,2) A2172	A14 Bionic (E)	8	256	128 KiB	64B
iPhone 11	A13 Bionic (P)	6	512	192 KiB	64B
(iPhone12,1) A2111	A13 Bionic (E)	6	256	96 KiB	64B
iPad Pro 11"	A12X (P)	8	256	128 KiB	64B
(iPad8,3) A2013	A12X (E)	6	128	48 KiB	64B
iPad 6th Gen (iPad7,5) A1893	A10X Fusion	4	256	64 KiB	64B
iPhone 7 Plus (iPhone9,4) A1784	A10 Fusion	4	256	64 KiB	64B

Table 3.2: The L1-D cache organization of various Apple CPUs. **W**: Ways, **S**: Sets, **CL**: Cache Line size.

Name	CPU	L1-D Cache			
		W	S	Size	CL
MacBook Air (M2, 2022)	M2 (P)	8	256	128 KiB	64B
(Mac14,2) A2681	M2 (E)	8	128	64 KiB	64B
MacBook Pro 14"	M1 Max (P)	8	256	128 KiB	64B
(MacBookPro18,4) A2442	M1 Max (E)	8	128	64 KiB	64B
MacBook Pro 14"	M1 Pro (P)	8	256	128 KiB	64B
(MacBookPro18,3) A2442	M1 Pro (E)	8	128	64 KiB	64B
Mac Mini (M1, 2020)	M1 (P)	8	256	128 KiB	64B
(Macmini9,1) A2348	M1 (E)	8	128	64 KiB	64B
iPhone 13 mini	A15 Bionic (P)	8	256	128 KiB	64B
(iPhone14,4) A2481	A15 Bionic (E)	8	128	64 KiB	64B
iPhone 12	A14 Bionic (P)	8	256	128 KiB	64B
(iPhone13,2) A2172	A14 Bionic (E)	8	128	64 KiB	64B
iPhone 11	A13 Bionic (P)	8	256	128 KiB	64B
(iPhone12,1) A2111	A13 Bionic (E)	6	128	48 KiB	64B
iPad Pro 11"	A12X (P)	8	256	128 KiB	64B
(iPad8,3) A2013	A12X (E)	4	128	32 KiB	64B
iPad 6th Gen (iPad7,5) A1893	A10X Fusion	4	256	64 KiB	64B
iPhone 7 Plus (iPhone9,4) A1784	A10 Fusion	4	256	64 KiB	64B

Table 3.3: The L2 cache organization of various Apple CPUs. **W**: Ways, **S**: Sets, **CL**: Cache Line size. **-**: the pLRU-based minimal eviction set algorithm did not resolve, and we could not confirm using an alternative information source. **†**: these devices use a random replacement policy for the L1 cache [88], precluding us from using the eviction set algorithm. *****: we could not confirm the results, but these are the most likely based on other Apple CPUs with similar cache organization.

Name	CPU	L2 Cache			
		W	S	Size	CL
MacBook Air (M2, 2022)	M2 (P)	16	8192	16 MiB	128B
(Mac14,2) A2681	M2 (E)	16*	2048*	4 MiB	128B
MacBook Pro 14"	M1 Max (P)	12	8192	12 MiB	128B
(MacBookPro18,4) A2442	M1 Max (E)	16*	2048*	4 MiB	128B
MacBook Pro 14"	M1 Pro (P)	12	8192	12 MiB	128B
(MacBookPro18,3) A2442	M1 Pro (E)	16*	2048*	4 MiB	128B
Mac Mini (M1, 2020)	M1 (P)	12	8192	12 MiB	128B
(Macmini9,1) A2348	M1 (E)	16	2048	4 MiB	128B
iPhone 13 mini	A15 Bionic (P)	12*	8192*	12 MiB	128B
(iPhone14,4) A2481	A15 Bionic (E)	16*	2048*	4 MiB	128B
iPhone 12	A14 Bionic (P)	16	4096	8 MiB	128B
(iPhone13,2) A2172	A14 Bionic (E)	16*	2048*	4 MiB	128B
iPhone 11	A13 Bionic (P)	16	4096	8 MiB	128B
(iPhone12,1) A2111	A13 Bionic (E)	16*	2048*	4 MiB	128B
iPad Pro 11"	A12X (P)	16	4096	8 MiB	128B
(iPad8,3) A2013	A12X (E)	-	-	2 MiB	128B
iPad 6th Gen (iPad7,5) A1893	A10X Fusion	†	†	3 MiB	128B
iPhone 7 Plus (iPhone9,4) A1784	A10 Fusion	†	†	3 MiB	128B

We hypothesize that speculated branches will resolve much quicker when the data is in the L1 cache than the L2 cache or main memory.

Measuring L1 and L2 Speculation Depth. We use the cycle-accurate timer described in subsection 3.2.5 to evict a series of values determining the condition of a branch instruction, thereby opening a speculation window. At the branch target, we introduce a sequence of data-dependent `mul` instructions followed by a load instruction for an uncached probe element. Here, we hypothesize that if we introduce enough `mul` instructions, the speculation window will be too short to reach the load instruction when the data is in either the L1 or L2 cache. Accordingly, we conduct an experiment where we evict the branch condition from the L1 and L2 caches, execute the branch, and measure the load latency to the probe element to determine how many `mul` instructions we need to introduce. We perform 1,000 trials for each number of `mul` instructions.

Results. Figure 3.3 shows the number of `mul` instructions that we can introduce before the probe element remains uncached. While each platform is different, we see that Apple’s M series CPUs can execute about 100 `mul` instructions under speculation. With each `mul` requiring 3 cycles [89], this translates to speculative windows of about 300 instructions on high-end Apple platforms.

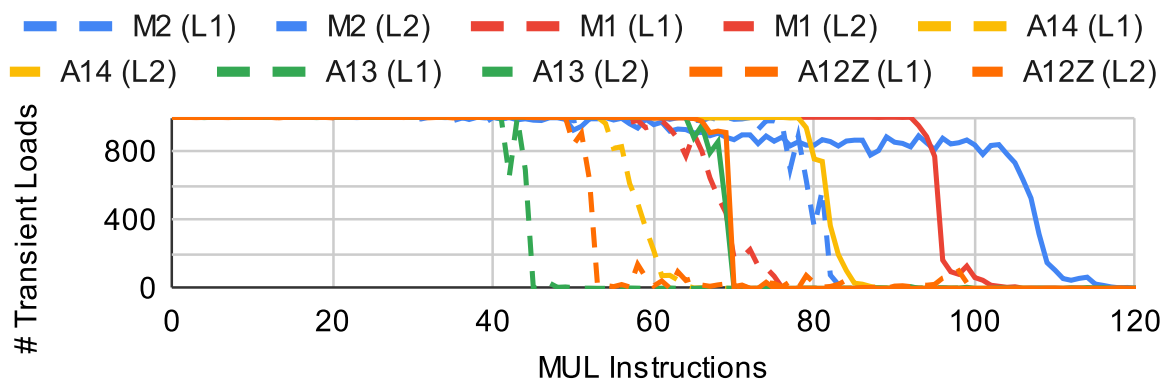


Figure 3.3: The number of `mul` instructions that can be executed speculatively when the condition is evicted from the L1 cache (dashed) vs. the L2 cache (solid) on various CPUs.

3.4.3 [P₃]: Discerning Cache Hits From Misses

In this section, we consider the problem of discerning cache hits from misses in timer-restricted environments. Recall that in non-privileged environments, we cannot rely on high-resolution timers to achieve this for individual addresses. A solution to this problem is also suitable for other environments with restricted timer resolution, such as all web browsers (see section 3.5), and is thus of independent interest. We first describe a primitive to discern cache hits from misses at each cache level, using speculation to make measurement possible with only low-resolution timers. We further extend this gadget to a timerless variant, which is made possible by race conditions instead of timing data.

Constructing a Distinguisher Using Speculation. We now combine the speculation depth information for each cache level from subsection 3.4.2 with the pLRU gadget as described in subsection 3.2.6 to construct a distinguisher gadget that can tell whether the data for a given address is present or not in a specific level (L1 or L2) of the cache hierarchy. The distinguisher gadget resembles our setup for measuring speculation depth. More specifically, it speculates on a conditional branch, where this time the condition is the target data to be measured. The branch target begins with the number of `mul` instructions corresponding to the cache level to measure, where this number is derived from our results in Figure 3.3. Afterwards, instead of loading an uncached probe element, the gadget transmits a 1 using the pLRU gadget. That is, a 1 will be transmitted only if the target data is not present in the cache level being measured.

Gadget Overview. Listing 3.2 is the pseudocode of our distinguisher gadget. After initializing the pLRU channel (Line 1), we branch on the value of the address to which `target` points (Line 2). If the CPU reaches Line 6 under speculation, the pLRU gadget transmits a 1, indicating a cache miss. Otherwise, the address to which `target` points is a cache hit, and no transmission occurs because speculation ends before Line 6.

Next, if a low-resolution timer is available, we can use it to measure the execution time of `plru.traverse()` in an attempt to ascertain whether a transmission had occurred

Listing 3.2: Our speculation-based gadget to distinguish cache hits from misses in the absence of high-resolution timers.

```

1 plru.init()
2 if (*target != 0) {
3     x *= 1;
4     ... // target cached - speculation ends here
5     x *= 1;
6     plru.transmit(1);
7     // target not cached - speculation ends here
8 }

```

inside our distinguisher gadget:

```

is_cache_miss =
    badtimer.time(plru.traverse()) > threshold;

```

While this is similar to `plru.receive` from Listing 3.1 in that we time `plru.traverse` with a low-resolution timer against a coarse-grained threshold, we note that the resulting value now directly corresponds to the target's cache state. That is, `plru.traverse`'s runtime only exceeds `threshold` in case Line 6 was speculatively executed due to a prolonged speculation window induced by a cache miss in Line 2.

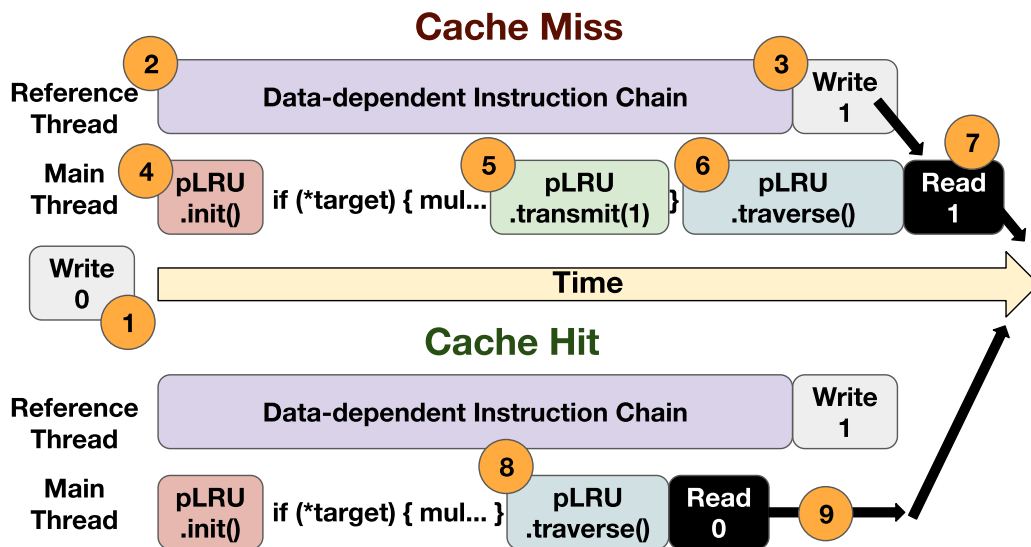


Figure 3.4: An overview of how our race condition-based gadget runs to distinguish cache hits from cache misses.

Measuring Without Timers via Race Conditions. We recall from subsection 3.2.6 that `plru.traverse` allows one to arbitrarily amplify L1 miss latencies, making them visible using low-resolution timers. We now use this amplification property to remove the need for a timer altogether, allowing our gadgets to operate in a timerless environment. At a high level, we race two threads that update a shared variable, such that we can determine the order in which the threads finished running. Figure 3.4 outlines our gadget, where the syntax `#` is used to describe each part of the figure in detail.

Our race consists of a zero-initialized shared variable `①`, a reference thread, and a main thread. The reference thread `②` always executes a chain of data-dependent instructions, and then writes 1 `③` to the shared variable. The main thread `④` starts concurrently, runs the distinguisher gadget in Listing 3.2, calls `plru.traverse`, and then reads the shared variable, outputting the result.

We now argue that after calibrating `plru.traverse`, the output of the main thread in fact corresponds to the cache state of the target variable. More specifically, if the target data is not cached, the CPU has a long speculation window for our distinguisher gadget in Listing 3.2, reaching `plru.transmit(1)` `⑤` under speculation. This causes the subsequent `plru.traverse` routine `⑥` to take longer, finishing after the reference thread has written 1. Therefore, the main thread reads 1 `⑦` in this case.

In contrast, if the target data is cached, the CPU only has a short speculation window in Listing 3.2, never reaching the `plru.transmit(1)` operation. Next, as nothing is transmitted over the pLRU channel, the `plru.traverse` routine terminates quickly `⑧`. With the race results being flipped, this causes the main thread to read 0 before the reference thread can write 1. `⑨`

Experimental Results. We now evaluate how accurately the distinguisher gadget classifies four outcomes: hits and misses, on the L1 and L2 caches. Table 3.4 summarizes our findings, as we degrade the timer resolution on an Apple M1 machine from an unprivileged 42 ns timer to removing the timer altogether. As can be seen, we observe perfect discern-

ment even with a timer resolution of 1 ms. The timerless version likewise achieves high accuracy, particularly for distinguishing L2 hits from misses.

Table 3.4: Probability of a correct observation (out of 1000 runs) by our distinguisher gadget, across various timer resolutions.

Timer Resolution	42 ns	10 μs	100 μs	1 ms	Timerless
L1 Hit	100%	100%	100%	100%	82%
L1 Miss	100%	100%	100%	100%	77%
L2 Hit	100%	100%	100%	100%	99%
L2 Miss	100%	100%	100%	100%	97%

3.4.4 $[P_4]$: Constructing Minimal Eviction Sets

Now, we use our distinguisher gadget to find minimal L2 eviction sets from an unprivileged user, even in environments that only provide low-resolution timers, or no timer at all. As browser-based code does not have access to any cache flush instructions, finding minimal eviction sets in timer-restricted environments is a common side-channel task, making this technique of independent interest.

Baseline Approach of [25]. Eviction set finding begins with an inflate step, where we keep allocating and accessing elements until the victim address is evicted. Its output is called a *conflict set*, and the goal is to reduce it to a minimal eviction set (i.e., whose size is the cache’s associativity a). Vila et al. [25] use group testing to expedite reduction, assuming a is known. In each iteration, their algorithm divides the conflict set into $a + 1$ bins, withholds one bin at a time, and checks if the remaining bins still evict the victim. If so, the withheld bin is discarded, and the remaining bins become the conflict set for the next iteration. Due to the Pigeonhole principle, there are ≥ 1 bins that can be discarded each iteration from a conflict set of size n . Hence, the algorithm discards $\frac{n}{a+1}$ elements per iteration until $n = a$.

Improving the Reduction. We improve this technique by generalizing it to operate on an upper bound k of a , rather than the exact number of ways. Then, during each iteration,

we divide the conflict set into k bins instead of $a + 1$ bins. We deviate from [25] by testing the remaining bins after discarding a bin, instead of starting the next iteration. This lets the next iteration start with the minimal set of bins required for eviction. We note that in the case k is large (i.e., 64), most bins are redundant. Thus, our reduction removes $k - a$ bins on average per iteration, instead of just one.

Improving the Backtracking. Noise sometimes causes the reduction step to remove bins that are essential for eviction. [25] remediates this with a backtracking step that adds the last removed bin back to the conflict set. However, we found this approach is still susceptible to noise, especially when used with the speculation-based distinguisher gadget from subsection 3.4.3, and often the algorithm fails to converge. We improve this by reallocating as many of the previous elements until the conflict set evicts the victim again.

Empirical Results. We first implement both [25] and our algorithm using the privileged cycle-accurate timer to time the victim’s load latency as a baseline, and measure the time to convergence and success probability over 20 trials on an Apple M1. Here, we define success as the resulting eviction set being minimal and able to evict the victim. We then introduce the distinguisher gadget and use it alongside coarser timing sources, starting with the unprivileged timer in macOS with 42ns resolution and ending at no timer (for the latter, we use the variant with race conditions). We present the results in Figure 3.5. While our reduction algorithm’s time to convergence is longer, it eliminates false positives (i.e., when the reduction converges but the resulting eviction set fails to evict the victim) for all timer resolutions except when the timer is removed.

3.4.5 $[P_5]$: Testing for L2 Inclusiveness

With the ability to efficiently and reliably find minimal L2 eviction sets, we now ascertain that the L2 cache is *inclusive*. Otherwise, the L2 cache is *non-inclusive* or *exclusive*. We test for inclusiveness using an L2 eviction set and the distinguisher gadget from subsection 3.4.3. We check if traversing the L2 (shared per-cluster) eviction set leads to L1

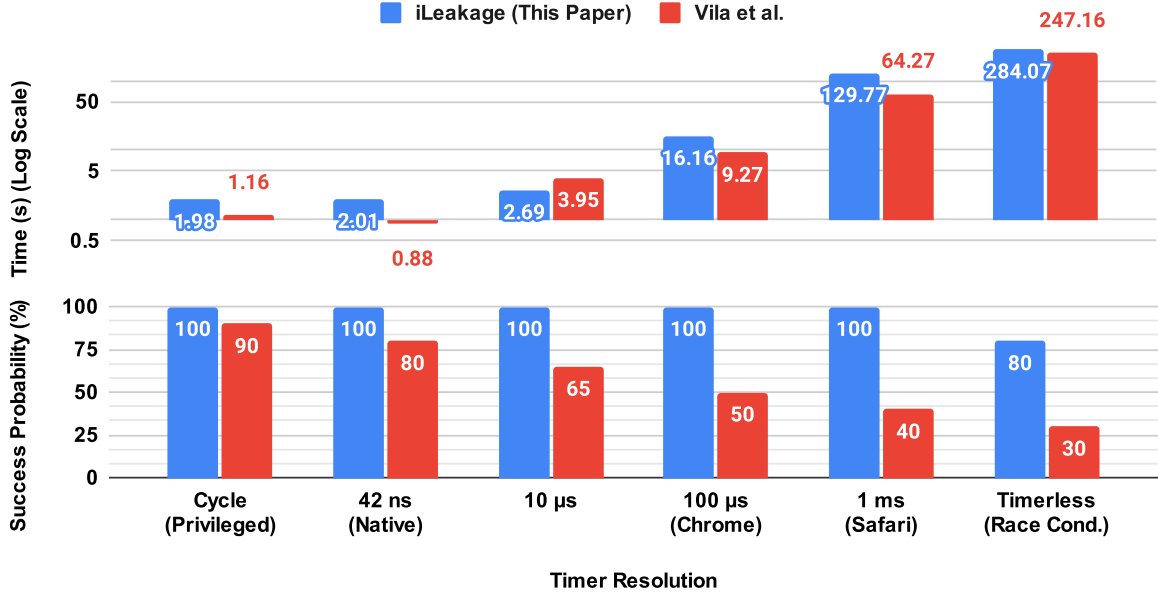


Figure 3.5: Comparison of eviction set-finding algorithms (ours and [25]) across degrading timer resolutions.

(private) cache evictions in another core. That is, we expect to observe cross-core L1 evictions only if the cache is inclusive. Indeed, we find that the L2 cache of the Apple M1 is inclusive of the L1 caches. Furthermore, given that the L2 cache line size (128 B) is double that of L1 (64 B), we observe improved eviction rates when accessing both halves of each L2 cache line.

Experimental Setup. Noticing the discrepancy in cache line sizes between the L1 and L2 caches, we design experiments to determine if this affects their inclusiveness. In our first three experiments, we access the whole L2 cache line, the first 64 bytes, and the last 64 bytes respectively during the eviction test, and measure the L1 eviction rate of the victim (which is cached in another core’s L1). As a control, our fourth experiment evicts an unrelated L2 cache set, which should leave the victim’s L2 cache line intact. Subsequently, we measure the effect of repeated eviction set traversals. We use a spinlock to synchronize the attacker’s thread with the victim thread, wherein the attacker’s thread shuffles and traverses the eviction set 1, 2, or 3 times before yielding to the victim thread.

Results. We show the cross-core L1 eviction rates in Figure 3.6. Here, we observe from

the first three columns that while accessing half of an L2 cache line occasionally results in evictions, accessing both halves approximately triples the success rate. We also verify that evicting a different L2 cache set does not result in victim evictions. For the experiments with synchronization, we observe that accessing the eviction set more often results in higher eviction rates, with 3 access iterations guaranteeing eviction. Overall, our results strongly indicate that the M1’s L2 cache is inclusive.

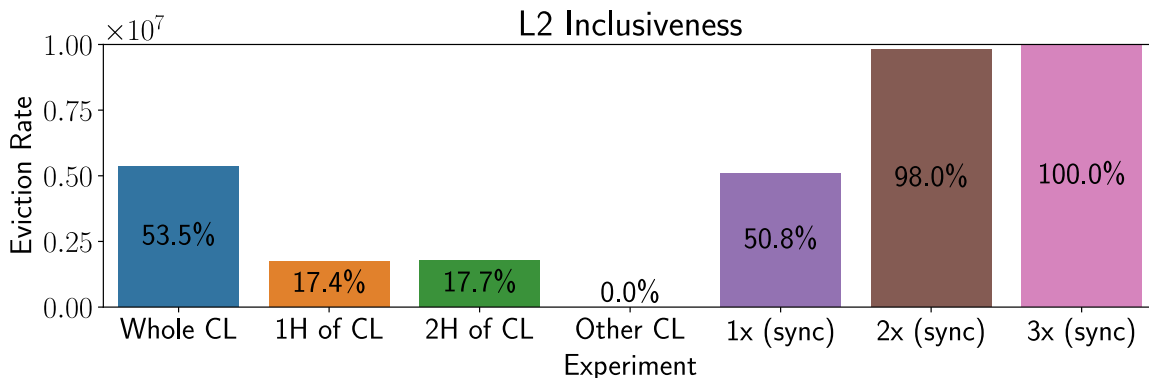


Figure 3.6: L1 cache eviction rate of a victim when evicting its L2 cache line from another core on the Apple M1 CPU.

3.4.6 $[P_6]$: Timerless Spectre Attacks

We recall from subsection 3.2.5 that all major web browsers have degraded their timer resolution in an attempt to harden browsers against side-channel attacks [47, 48, 49]. In fact, browser vendors has gone as far as to restrict the use of `SharedArrayBuffers`, in an effort to stop attacks from building their own timers [50]. While concurrency and shared state continues to be available through web workers and the message passing API [26] due to compatibility reasons, the latency of these interfaces has been artificially increased to be orders of magnitude greater than that needed to distinguish a single cache miss from a hit.

Building on our timerless attack primitives presented earlier, we are able to use the browser’s message passing API to cause race conditions, allowing us to execute the cache-distinguisher method from subsection 3.4.3. We then construct a timerless browser-based Spectre v1 gadget, using race conditions rather than cache timing to recover the secrets

leaked during speculative execution.

Table 3.5 shows accuracy and leak rate for a proof-of-concept timerless Spectre-v1 attack using this approach, benchmarked on an Apple M1 laptop running unmodified Safari 16.2, Firefox 108.0, and Tor Browser 12.0.1 (all latest at the time of writing). While the attack’s leak rate is relatively low due to noise introduced by the JavaScript environment, we note the attack achieves close-to-perfect accuracy without any explicit or constructed timers.

Table 3.5: Timerless Spectre-v1 performance across browsers.

Metric	Tor Browser	Firefox	Safari
Accuracy	96.15 %	98.79 %	97.46 %
Leak Rate	10.63 b/s	10.76 b/s	8.26 b/s

3.5 Attacking the Safari Browser

We now present iLeakage, a JavaScript-based transient-execution attack that recovers secret information from the Safari browser. In addition to bypassing standard side-channel countermeasures (such as a low-resolution timers) deployed by all browser vendors, iLeakage overcomes several Safari- and Apple-specific challenges.

[C₁] Site Isolation. Just like Chrome and Firefox, modern versions of Safari attempt to prevent mutually-distrusting webpages from using the same rendering process, compartmentalizing different websites into different address spaces.

[C₂] 35-bit Pointers and Value Poisoning. In addition to site isolation, Safari also limits the attacker’s ability to craft and dereference arbitrary 64-bit pointers. All indexing in JavaScript objects is 32-bits and most object accesses entail 35-bit compressed pointers, while 64-bit doubles are poisoned. Thus, an attacker must bypass Apple’s isolation countermeasures to retrieve sensitive information.

[C₃] Obtaining Deep Speculation. To overcome [C₂], we use speculative type confusion that requires the CPU to speculate past multiple condition checks. Thus, we need a

consistent method to ensure that the CPU does not revert the speculation before the attack completes executing transiently and leaks the sensitive data.

[C₄] Reliability. Finally, we design a reliable exploit that can be used to leak data multiple times. In particular, we need to architecturally hide all the type confusion events to prevent the browser from raising exceptions or falling back to the JavaScript interpreter.

Attack Overview. We first resolve [C₁] by using the `window.open` JavaScript function, observing that it brings the target website’s data into the attacker’s address space in Safari. For [C₂], we forge 64-bit pointers around value poisoning countermeasures via Safari’s performance optimization within its mitigations for architectural type confusion. We show this mitigation is insufficient for speculative type confusion, with the CPU transiently dereferencing our forged pointer. For [C₃], we inspect Safari’s memory allocator and data structures for JavaScript objects to selectively evict the type of our attacker object from the L2 cache, but keep the rest of the object in the L1-D cache. Finally, for [C₄], we hide the type mismatch events from Safari with another layer of speculation.

3.5.1 [C₁]: Bypassing Process Isolation

To mount a speculative execution attack, the attacker must coerce the target webpage into its address space. Recognizing this, both Chrome and Firefox recently implemented a Site Isolation paradigm [90, 91] to ensure that different rendering processes handle pages that have different origins and are not subdomains of the same parent domain (e.g., `a.site.com` and `b.site.com`).

Safari’s Isolation Model. Taking this approach a step further, Safari follows a simple one process per tab model, where two webpages are never consolidated into the same rendering process, even under high memory pressure and even if they share an eTLD+1 in their Uniform Resource Locator (URL)s. Instead, Safari spawns a new rendering process for each tab until the system runs out of memory. Empirically confirming this, we used a MacBook Air with an M1 CPU and 16 GB of Random Access Memory (RAM) and opened

177 tabs. While this had the effect of Safari refusing to open additional tabs, it never consolidated webpages into rendering processes, maintaining its one process per tab model. Similar results were obtained on an iPad Pro (12.9-inch, 5th generation).

Abusing `window.open` to Achieve Consolidation. Despite newly opened tabs failing to consolidate, we found that we can reliably render a target page inside the address space of an attacker’s page by using the `window.open` JavaScript API. In particular, `attacker.com` can call `window.open` to open a pop-up window rendering `target.com`. Crucially, while both websites appear in different windows, Safari uses a single rendering process for both pages, causing both websites to end up in the same address space. Notably, websites cannot refuse `window.open` to render them in a separate window, making this technique applicable for any target website. Finally, while we also observe cross-origin iframes consolidating, WebKit’s Intelligent Tracking Prevention countermeasure prevents the delivery of cookies to cross-origin iframes [92], precluding them from rendering secrets.

Unavailability of `SharedArrayBuffer`. We recall from subsection 3.2.5 that the `SharedArrayBuffer` API can be used to build high-resolution timers. However, to mitigate side channels, all major browsers now limit its availability to cross-origin isolated pages. Notably, we observe that cross-origin isolated webpages fail to consolidate in Safari with any other webpage using either technique. In turn, this implies we cannot use `SharedArrayBuffer` to craft a high-resolution timer for our attack. Accordingly, we use the pLRU covert channel described in subsection 3.2.6.

3.5.2 [C₂]: Speculative Type Confusion

Despite being a 64-bit application, for increased memory safety, Safari uses 32-bit array indices and 35-bit pointers to the underlying storage of most objects, partitioning them into 32 GB compartments named Gigacages. As secrets in a webpage’s Document Object Model (DOM) are located outside this Gigacage, they remain out of reach for naïve Spectre

attackers that can only corrupt 32-bit indices or 35-bit pointers.

In this section we overcome this countermeasure, building a speculative type confusion attack which can read arbitrary 64-bit addresses in the page’s rendering process. While we acknowledge prior type confusion techniques against Chrome [31, 46, 93] and the Linux kernel [94], to the best of our knowledge this is the first application of speculative type confusion to Apple’s ecosystem in general and the Safari web browser in particular.

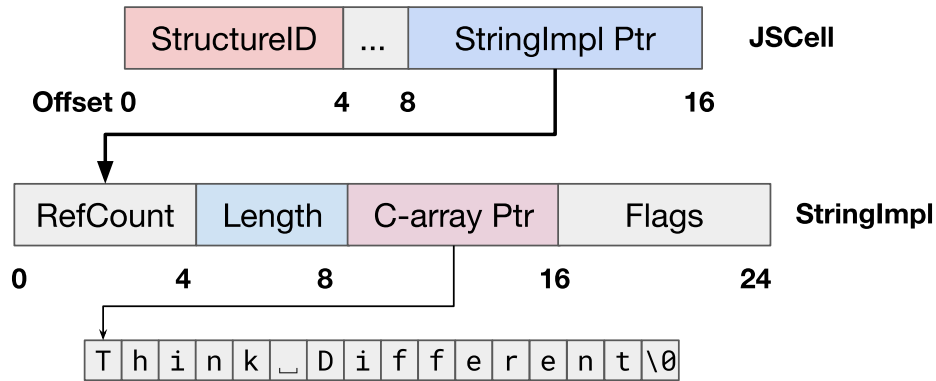


Figure 3.7: Memory layout of JavaScript strings in WebKit. The `JSCell` class is common to all objects, while the `StringImpl` class is the backing store specific to strings.

Locating 64-bit Pointers. We begin our investigation of speculative type confusion attacks in Safari by inspecting the memory layout of common JavaScript objects, paying special attention to 64-bit pointers. A particularly convenient object containing a 64-bit pointer is JavaScript’s `string` object, whose memory layout we show in Figure 3.7. As can be seen in Figure 3.7 (bottom), WebKit writes characters in a separate C-style array, and dereferences a 64-bit C-array pointer to access it (Figure 3.7 (middle), `StringImpl`).

Performing a String Indexing Operation. More specifically, Listing 3.3 contains pseudocode of JavaScript’s string indexing operation, when compiled by WebKit’s JIT compiler. The code takes as input a variable `inp`, a WebKit metadata object called `JSCell`, common to all JavaScript objects. Having verified that the type inside `inp` indeed corresponds to a string (Line 3), Listing 3.3 proceeds to retrieve the corresponding `StringImpl` structure (Line 6), a WebKit internal structure holding the string’s length and a 64-bit pointer to the string’s actual content. After checking that the index being referenced is smaller

than the string's actual length (Line 8), Listing 3.3 retrieves the corresponding character by dereferencing the 64-bit `cArrayPtr` (Line 11).

Listing 3.3: JIT-compiled WebKit string indexing operation.

```
1 String::operator[] (inp, index) {
2     // Check StructureID first
3     if (inp.StructureID != String::StructureID)
4         exitToInterpreter();
5     // Dereference ptr to get StringImpl
6     StringImpl impl = *(inp.ptr);
7     // String length check
8     if (impl.length <= index)
9         exitToInterpreter();
10    // Dereference ptr to C-array
11    return *(impl.cArrayPtr + index);
12 }
```

Attacker Object Setup. We assume the attacker would like to dereference some 64-bit address `addr`. Then, the attacker has to create an object whose memory layout resembles that of a JavaScript string. As objects must support heterogeneous property types, WebKit implements this by making every property a `JSValue` class. Conveniently, each `JSValue` is 64 bits wide. However, in order to craft a `JSValue` whose value holds `addr`, the attacker needs control over all 64 bits. This is not trivial, as some of `JSValue`'s 64 bits are patterned by WebKit to indicate which property type it holds. Aside from references to other JavaScript objects (which are always the attacker's own data), an attacker can only fill `JSValue` with integers, floats, `True`, `False`, `Undefined`, and `null` values. Figure 3.8 indicates the bit pattern corresponding to each value.

Value Poisoning via NaN-boxing. Using bit patterns to indicate different property types within a fixed-width datatype is a technique known as NaN-boxing. Remarkably, in WebKit's implementation of NaN-boxing, bits 63-48 are non-zero for integers. Furthermore, WebKit uses the IEEE 754 double-precision format [95] to represent floats. While this format requires control over all 64 bits, WebKit adds a poison value of 2^{49} to the encoded bits, making bit 49 or higher always set in memory. In both cases, the 64 bits comprising

memory layout of a string. At the `JSCells` for the two objects, we note the pointers to the underlying storages (`StringImpl Ptr` and `Butterfly Ptr`) are at an offset of eight bytes in both cases. In the underlying storage containing an array of floats, Line 3 of Figure 3.9 (top) puts the value `0xffff` at offset 4 (after IEEE 754 conversion) where the string length resides, and line 5 puts the target address `addr` at offset 8, which contains the C-array pointer in the string.

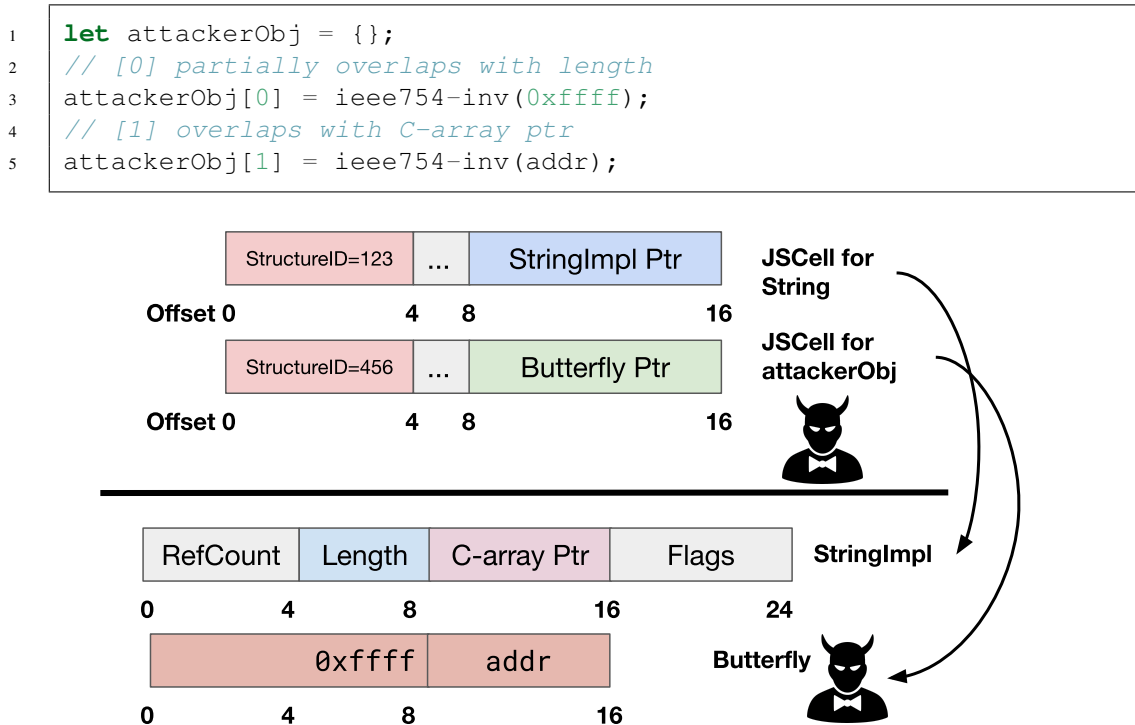


Figure 3.9: (Top) Pseudocode to construct a malicious object for transiently dereferencing the target address `addr`. (Bottom) A comparison with the memory layout of a `string` object.

Dereferencing Arbitrary 64-bit Pointers via Speculative Type Confusion. To dereference the 64-bit pointer, we consider the case where the code in Listing 3.3 is executed on the `attackerObj` created in Figure 3.9 (top) with `index=0`. Before this code executes, we evict the `StructureID` of our object from the cache to delay the resolution of the branch at Line 3. This essentially forces the CPU to speculate forward, dereferencing the object’s butterfly pointer and treating the resulting butterfly as a `StringImpl` structure (Line 6). Next, the CPU incorrectly uses the data in `attackerObj[0]` as the string

length (Line 9). As the string length is `0xffff`, which is larger than `index`, the CPU bypasses the branch and incorrectly uses the data in `attackerObj[1]` as the string's `cArrayPtr`, dereferences it and returns the resulting value under speculation.

Type Eviction. To ensure the type confusion attack succeeds, we delay the resolution of the branch in Line 3 of Listing 3.3 as much as possible by evicting the cache line holding the `StructureID`. However, this poses two challenges. First, the attacker must be able to construct eviction sets for a given address, overcoming Safari's 1 ms timer. Next, the attacker must keep the rest of the data of the `attackerObj` cached to prevent the CPU's pipeline from stalling. Overall, to meet both requirements, we must force WebKit's internal memory allocator to place the `attackerObj` across cache line boundaries at particular offsets.

3.5.3 [C₃]: Partial Object Eviction

As outlined above, our type confusion primitives requires placing an `attackerObj` across cache line boundaries, such that we can evict its `StructureID` variable while retaining the rest of the object in the CPU's cache. Next, to evict the `StructureID`, we must find a methodology for evicting specific addresses from the CPU's cache. This implies bypassing Safari's 1 ms timer, in order to distinguish cache misses from cache hits.

Understanding Safari's Memory Allocator. We depict the ideal scenario from above in Figure 3.10. Unfortunately, we observe that WebKit's memory allocator always places JavaScript objects, such as `attackerObj`, at 16-byte boundaries. Next, we note that most Intel and AMD CPUs use 64-byte cache lines while recent Apple CPUs use 128-byte cache lines in their L2 cache. Thus, we cannot evict the `attackerObj`'s `StructureID` without also evicting its butterfly pointer, preventing our attack.

Constructing Split Objects. Inspecting WebKit's memory allocator, we find a performance optimization where for certain built-in objects in the JavaScript API, the first 8 instances for each object type are 'half-aligned', using 8-byte alignment boundaries.

While Safari’s 1 ms timer prevents the attacker from distinguishing cache hits from misses, thus constructing eviction sets, we recall the technique from subsection 3.4.4 for constructing eviction sets using only low-resolution timers. Porting a similar approach to Safari, we can evict `StructureIDs` of these objects, thereby facilitating our attack.

3.5.4 [C₄]: Speculative Suppression

Having demonstrated how to achieve type confusion between JavaScript’s `Intl.Locale` and `String` objects while making the `Intl.Locale`’s `JSCell` object straddle two cache lines, we now describe our memory read primitive end-to-end in Listing 3.4.

Listing 3.4: Our speculative type confusion primitive.

```
1  let malObj = new Intl.Locale("en-US");
2
3  for (let i = 0; i < 10000; i++)
4      gadget(0, 0, "training");
5
6  const junk = malObj[1];
7  malObj = [ieee754-inv(0xffff), ieee754-inv(addr)];
8  evict_type(malObj);
9  plru.init();
10 gadget(0xffff, index, malObj);
11 return plru.receive();
12
13 function gadget(condVar, index, confusionObj) {
14     if (condVar < confusionObj.length)
15         let val = confusionObj[index];
16         plru.transmit(val);
17 }
```

Setup. As our attack relies on speculatively performing type confusion between `Intl.Locale` and `String` objects, Line 1 allocates the `malObj` of type `Intl.Locale`, whose `JSCell` is split between cache lines as outlined in Figure 3.11.

Training. We call `gadget()` 10,000 times (Lines 3 – 5) with `condVar` and `index` set to 0, and `confusionObj` set to “training” to train the branch predictor. As `condVar` is less than 8, the CPU executes the branch, which sets `val` to ‘t’ and transmits ‘t’ over the pLRU channel (Line 14 – 16). By passing a string to the `gadget` function repeatedly,

Safari consequently specializes the access at Line 15 to use the Just In Time (JIT)-compiled code from Listing 3.3 instead of processing Line 15 with its JavaScript interpreter.

Attack Phase. We then ensure that the CPU cache contains our malicious `Intl.Locale` object (Line 6). We set the two indexed properties of `malObj` to be a fake string length and the address of the contents we wish to leak (Line 7).¹ As `malObj` is split between two cache lines, we can evict its `StructureID` while keeping its butterfly pointer cached (See subsection 3.5.3), which we do on Line 8. We then proceed to call `gadget()` on `malObj`.

Speculative Type Confusion. With the `StructureID` evicted, the CPU fails to retrieve the `length` property (Line 14), leading it to speculate the `if` as a consequence of mistraining. As the code at Line 15 is specialized to use Listing 3.3, calling `gadget()` on `malObj` results in a type confusion where the CPU dereferences and returns the value of `addr`. We then transmit (Line 16) this value over the pLRU channel.

Speculative Suppression. As Line 15 was executed speculatively, JavaScript cannot retrieve the `length` property of `malObj` while `StructureID` is evicted. Once the value of `malObj.length` is architecturally available however, the CPU rolls back the incorrect speculation at Lines 15 – 16. Thus, speculative suppression prevents Safari from de-specializing the memory access at Line 15 or producing a type error event, allowing us to keep on repeating the attack across multiple addresses.

Value Recovery. Finally, we have to retrieve the value stored at `addr`, which was transmitted through the pLRU channel (Line 16 of Listing 3.4). Here, we simply rely on the pLRU channel for this (Line 11), calibrating it for Safari’s 1 ms timer, see subsection 3.2.6.

3.5.5 End-to-End Attack Evaluation

Having described the collection of techniques to build our attack, we now proceed to evaluate its leakage rate and accuracy across a variety of Apple devices. More specifically, we

¹Recall that these values must be encoded as IEEE-754 floats, as explained in Figure 3.9.

run iLeakage on unmodified Safari and try to read a known 512-bit string, averaging accuracy over 10 trials. Table 3.6 contains a summary of our findings, showing that our attack can read arbitrary address at a rate of 24 to 34 bits per second, with an accuracy of 90% to 99%.

Table 3.6: End-to-end performance of iLeakage.

Device	Apple CPU	Leak Rate	Accuracy
iPad Pro 11" 2nd Gen.	A12Z Bionic	23.22 b/s	97.36 %
iPhone 11	A13 Bionic	28.89 b/s	99.18 %
iPhone 12	A14 Bionic	33.13 b/s	98.96 %
iPad Pro 12.9" 5th Gen.	M1	34.78 b/s	99.57 %
iPhone 13	A15 Bionic	26.29 b/s	95.92 %
MacBook Air (M1, 2020)	M1	32.97 b/s	98.48 %
MacBook Pro (14", 2021)	M1 Pro	33.28 b/s	90.96 %
MacBook Pro (14", 2021)	M1 Max	24.46 b/s	93.79 %
MacBook Air (M2, 2022)	M2	29.14 b/s	97.05 %

3.6 Weaponizing iLeakage

We now turn our attention to the implications of iLeakage on the security of the Safari web browser.

Experimental Setup. We use a MacBook Air (model A2337) with the M1 CPU and 16 GB of RAM for all attack experiments. We run Safari in an out-of-the-box configuration, with all side-channel countermeasures enabled.

Bringing Targets to Attackers. We begin by recalling that while Safari generally follows a strict process-per-tab model, pages opened by the `window.open` function share a rendering process with the parent page. Thus, we created an attacker page that binds `window.open` to an `onmouseover` event listener, allowing us to open any webpage in our address space whenever the target has their mouse cursor on the page. We note that even if the target closes the opened page, the contents in memory are not scrubbed immediately, allowing our attack to continue disclosing secrets. Finally, as `window.open` performs

consolidation regardless of the origins of both the parent and opened webpages, we host our attacker's page on a non-publicly accessible webserver, while using `window.open` to consolidate pages from other domains.

Attacking Gmail. With Google being one of the world's largest email providers, it is highly likely for a target to be signed in with their personal account. By having the event listener inside the attacker's page access execute `window.open(gmail.com)`, we can consolidate the target's inbox view into the attacker's address space. We then leak the contents of the target's inbox, see Figure 3.12.

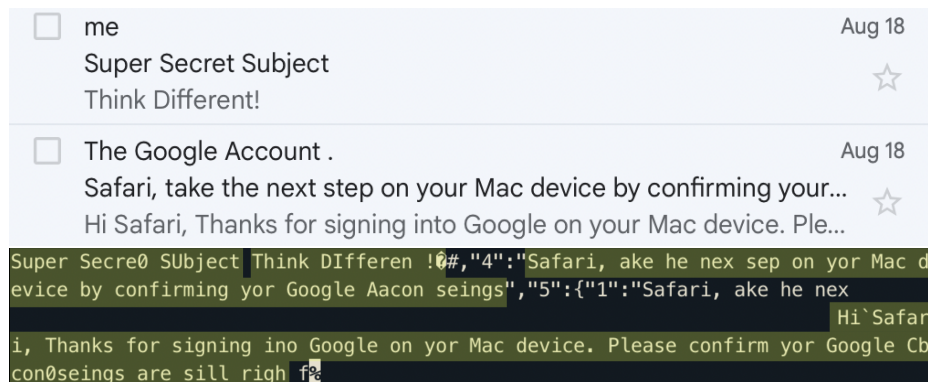


Figure 3.12: (Top) An email displayed in Gmail's web view. (Bottom) Recovered sender address, subject, and content.

Recovering Android Text Messages. Android users can send and receive text messages from a browser window by pairing their phone with Google's Messages platform. Thus, by opening Google Messages using `window.open()`, we can recover a target's text messages without attacking their mobile phone itself. See Figure 3.13.

IP Address and Geolocation. Finally, an attacker might decide to open a website the target does not normally visit, in order to learn more information about the targeted user. For example, in case the attacker does not have access to server logs for their malicious page (e.g., due to hosting on third party servers), the attacker can open an IP address geolocation page and subsequently recover the target's location, IP and ISP details. See Figure 3.14.

Attacking Password Managers. Going beyond reading website content, the popularity of password managers also allows us to go after login credentials of popular websites.

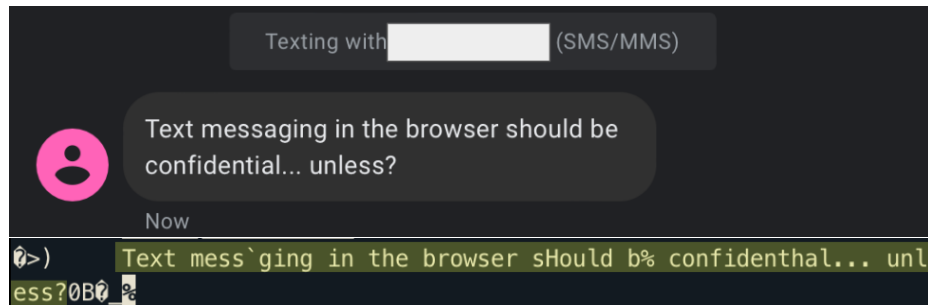


Figure 3.13: (Top) Text message sent to an Android phone which has been paired to the Google Messages webpage. (Bottom) Recovered text message in highlights.

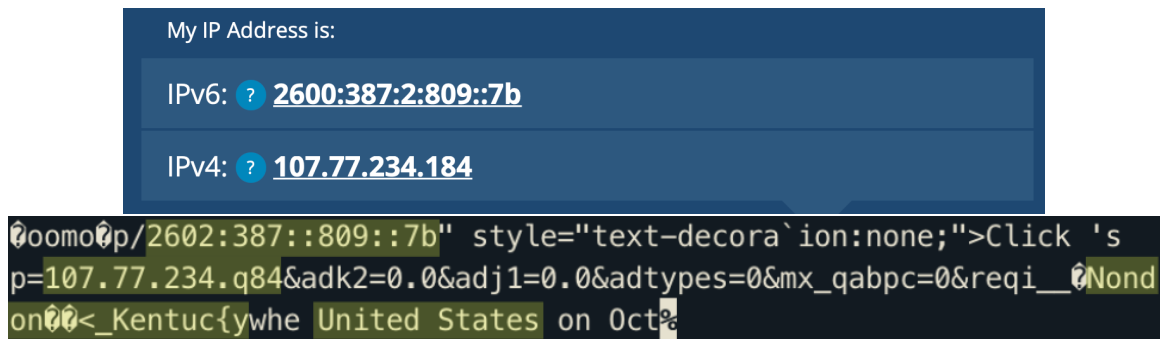


Figure 3.14: (Top) Website displaying the target's IPv4 and IPv6 addresses. (Bottom) Recovered information from the website's DOM in highlights, including geolocation.

More specifically, we installed LastPass version 4.107.1 (the latest at the time of writing) on our Safari browser. Next, while LastPass requires user interaction when autofilling credentials for the first time on a webpage, it automatically and permanently fills them in on subsequent logins without any interaction. Thus, by opening the login pages of popular websites (prompting LastPass to autofill credentials), we can recover the target's username and password. See Figure 3.15.

Forced Logout to Induce Login Page. To steal credentials, an attacker must first prompt a credential manager to autofill the target's password. Thus, when the target is already authenticated for a service, we must somehow log the target out to access the service's login screen subsequently, thereby triggering password autofills. Here, we observe that logout operations on most services do not require knowing any user-specific information or solving challenges (e.g., CAPTCHAs). Thus, an attacker can recover credentials from Amazon, GitHub, and Google by first sending an Asynchronous JavaScript and XML (AJAX)

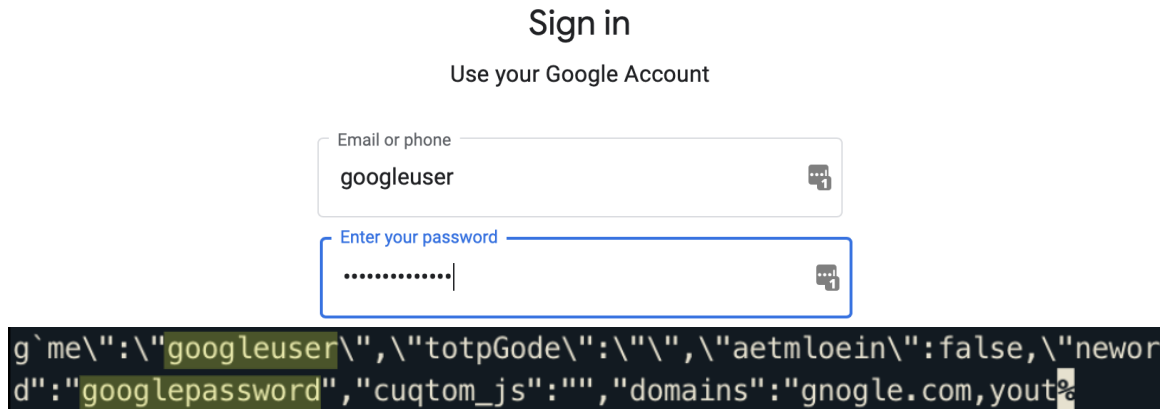


Figure 3.15: (Top) Google’s accounts page autofilled by LastPass, where the password is `googlepassword`. (Bottom) Leaked page data with credentials highlighted.

request in JavaScript to the logout endpoints of each service, then opening the login pages via our `window.open` technique.

3.7 Discussion

Observing the practicality of our end-to-end attack, we now discuss countermeasures to iLeakage, including Apple’s efforts during the responsible disclosure process. We also state iLeakage’s limitations and high-level takeaways from our attack primitives, some of which are applicable to other browser engines and vendors.

3.7.1 Countermeasures for Safari

Throughout section 3.5, our attack leveraged facets in Safari’s multi-process architecture and performance optimizations. We now discuss several countermeasure designs for closing these loopholes.

Preventing Consolidation via Site Isolation. Recall WebKit’s process model (subsection 3.5.1), where pages launched with `window.open` are consolidated into the address space of their parent’s rendering process. While this is sensible for real-world uses of `window.open` such as popups, as they need communication with their parent page, this design made our attack possible.

Following our disclosure, Apple developed a new Inter-Process Communication (IPC) API that makes spawning new processes for pages launched with `window.open` possible, in addition to pages opened in new tabs. We have empirically verified this mitigates our attack by preventing consolidation of domains across security boundaries. That is, while the speculative JavaScript sandbox escape is still possible, an attacker becomes limited to reading their own address space and therefore their own data. Finally, at the time of writing, Apple’s patch is publicly available [96] and is implemented in Safari Technology Preview versions 173 and newer.

Preventing Speculation Past Type Checks. In subsection 3.5.2 we constructed a 64-bit read primitive by confusing the CPU to assume the input is a string object, while in reality the input is an `Intl.Locale` object. WebKit’s JIT compiler can be hardened against type confusion attacks by inserting a fence instruction after every type check to prevent the CPU from speculating past it.

Removing Poisoning Optimizations. Also in subsection 3.5.2, we were able to fake a 64-bit pointer by abusing an optimization where WebKit would not poison floats in an array of floats, instead encoding that information in the array’s type information. As our attack demonstrates, this is risky in the face of speculative type confusion. Thus, while poisoning all floating-point numbers would not prevent speculative type confusion itself, it would prevent an attacker from reading the address space by using floats to craft 64-bit pointers. Yet, this comes with performance tradeoffs: every computation on a poisoned variable must be preceded by unpoisoning and succeeded by re-poisoning when writing the result back to memory.

3.7.2 Countermeasures for Websites

As full site isolation in Safari is under active development by Apple, we now discuss a countermeasure available to website administrators for mitigating our attack. Here, web pages can serve Cross-Origin Isolation (COI) Hypertext Transfer Protocol (HTTP) headers

with website endpoints containing secrets. We recall from subsection 3.2.5 and subsection 3.5.1 that COI was proposed as a Spectre countermeasure for websites, and pages with COI failed to consolidate in Safari.

COI consists of two HTTP headers: Cross-Origin Opener Policy (COOP) set to **same-origin**, and Cross-Origin Embedder Policy (COEP) set to **require-corp**. COOP states the current webpage must not be able to communicate with other pages, while COEP requires third-party resource to explicitly opt into being loaded on the current webpage [81]. In current release versions of Safari, we identify that the presence of the correct COOP header is alone for WebKit to spawn the webpage in a new process.

Measuring COI Adoption. Accordingly, we measure how widely popular websites have adopted COI or the COOP header. We take the Alexa top-100 websites, and also visit every URL linked from the main page of each website. On each URL, we check the HTTP headers across four user agents: {Safari 16.0, Chrome 105} for {macOS, iOS}. For both browsers, we crawled 3,058 URLs for macOS and 2,551 for iOS (some hyperlinks are not included in mobile versions of webpages) for a total of 5,609 URLs. Finally, our web crawler did not visit webpages that require authentication.

Results. At the time of writing, we find that only 113 of 5,609 URLs (2%) serve the COOP header with the correct value. Matching these 113 URLs to their corresponding Alexa top 100 websites, we find that 13 websites serve the COOP header at some page in their domain, thus resulting in a 13% adoption rate. Also, we find the user agent influences whether a server includes either header for some websites, depending on if the client is desktop or mobile. As such, we recommend more websites to opt-in such that Safari will never consolidate their webpages with another webpage.

3.7.3 Limitations

Leak Rate. As shown in Table 3.6, our attack recovers data at rate of about 30 bits per second. While our attack’s bottleneck is the transmission rate of pLRU covert channel with

Safari’s 1 ms timer, we do acknowledge our attack’s relatively low leakage. Thus, we leave designing high speed covert channels that are robust enough to use low resolution timers to future work.

Inability to Cross Address Spaces. Being a Spectre-style attack, iLeakage cannot read information present in other address spaces. While early versions of Apple hardware were susceptible to Meltdown [97], there is no indication of such vulnerabilities in newer Apple CPUs. We thus leave the task of exploring cross-address space attacks on modern Apple silicon to future work.

3.7.4 Broader Implications

We now discuss the components of our attack which are transferable to other platforms, and offer general takeaways for designing memory-safe browser engines in the face of speculative execution.

Transferability. As all iOS-based browsers are mandated to use WebKit as their underlying engine by Apple’s App Store policy, the end-to-end exploit chain of iLeakage affects all iOS browsers beyond Safari. Furthermore, beyond the Apple ecosystem, some of Samsung’s mobile and embedded devices use the Tizen operating system, whose default bundled browser uses WebKit. As the CPUs and overall architecture of Samsung devices is significantly different from Apple platforms considered in this chapter, we leave the task of investigating these devices to future work. Finally, our microarchitectural primitives, such as the timerless cache hit/miss distinguisher gadget using race conditions, are agnostic to browser engine, as we have shown with our timerless Spectre-v1 PoC on Firefox and Tor Browser in subsection 3.4.6.

Memory Safety Under Speculation. Both the Spectre-v1 PoCs and the end-to-end Safari exploit were made possible when assumptions about memory safety that hold true architecturally failed to hold under speculation. While it is well known that assuming variables will be in-bounds following a length check leads to Spectre-v1, we uncover more assumptions

and implementation details which warrant re-evaluation. More specifically, while Safari’s design is sufficient to prevent architectural type confusion, our work shows that speculative type confusion is still possible. Thus, we argue that speculative memory safety must be considered separately from architectural memory safety during system design.

Firstly, assuming objects will be of the correct type following a check may lead to sandbox escapes, even in the presence of standard Spectre countermeasures. Secondly, implementing optimizations or corner cases based on type may also become affected by misspeculations on type checks, even if they guarantee perfect security under architectural execution. Finally, making allocations such that attributes of objects which are essential to memory safety (e.g., type or length) map to a different cache line than the data of objects is risky, as the CPU can speculate past these safety checks when the object is partially evicted.

3.8 Conclusion

In this chapter, we study the side-channel resilience of recent Apple CPUs. Overcoming the lack of high-resolution timers in both native environments and JavaScript in the Safari web browser, we introduce primitives to mount cache attacks on degraded timers and notably without timers, after empirically measuring the cache organization, inclusiveness, and speculation depth. Furthermore, we show these primitives are transferable with Spectre-v1 proof-of-concepts in several browsers, and present algorithmic improvements for finding eviction sets with our new timing primitives. Subsequently, we show that numerous architectural invariants for memory safety in Safari’s design and implementation break under speculation, resulting in an end-to-end speculative type confusion primitive with arbitrary 64-bit read capabilities. We demonstrate the practicality and severity of this attack with popular real-world scenarios and targets. Finally, we suggest several mitigations and take-aways for web browser vendors.

CHAPTER 4

SLAP: DATA SPECULATION ATTACKS VIA LOAD ADDRESS PREDICTION ON APPLE SILICON

Since Spectre’s initial disclosure in 2018, the difficulty of mitigating speculative execution attacks completely in hardware has led to the proliferation of several new variants and attack surfaces in the past six years. Most of the progeny build on top of the original Spectre attack’s key insight, namely that CPUs can execute the wrong control flow transiently and disclose secrets through side-channel traces when attempting to alleviate control hazards, such as conditional or indirect branches and return statements.

In this chapter we go beyond (speculatively) affecting control flow, and present a new data speculation primitive that stems from microarchitectural optimizations designed to alleviate data hazards. More specifically, we show that Apple CPUs are equipped with a LAP. The LAP monitors past addresses from the same load instruction to speculatively load a predicted address, which may incorrectly point to secrets at rest (i.e., never architecturally read by the CPU). Once the secret is retrieved, the LAP allows for a large speculation window that suffices for an adversary to compute on the secret, such as leaking it over a covert channel.

We demonstrate the LAP’s presence on recent Apple CPUs, such as the M2, A15, and newer models. We then evaluate the LAP’s implications on security by showing its capabilities to read out-of-bounds, speculatively invoke rogue functions, break ASLR, and compromise the Safari web browser. Here, we leverage the LAP to disclose sensitive cross-site data (such as inbox content from Gmail) to a remote web-based adversary.

4.1 Introduction

From the turn of the decade, there is a pivotal change in the desktop computing market. Whereas x86 CPUs from Intel and AMD had dominated the heavyweight CPU market in the past, new lineages from Apple and Qualcomm using the ARM architecture are emerging in market share. Over the past few years, they have become formidable competitors to the x86-based landscape, bringing equivalent or often better performance at a fraction of the power consumption.

Another change brought about in recent computer technology is the rise of transient execution attacks (e.g., Spectre [1] and Meltdown [2]), which exploit speculative and out-of-order execution to leak information across security domains [1, 2, 14, 15, 16, 52, 63, 64, 65, 66, 67, 68, 73, 98]. That is, nearly all currently known Spectre variants rely on the CPU speculating on control hazards, encompassing if-statements, indirect jumps, returns, and loops, speculatively diverting the CPU's control flow into code gadgets benefitting the attacker. Finally, Spectre has impacted nearly all modern heavyweight CPU designs, including recent generations released by Intel, AMD, Apple and other vendors.

Next, to further increase performance, computer architects proposed speculating on data flows in addition to control flow, aiming to alleviate data hazards encountered during software execution [99, 100, 101, 102, 103, 104]. Here, rather than waiting for the hazard to resolve, the CPU attempts to predict the value of the data being accessed and proceeds execution of younger instructions using the predicted value. While some forms of such data speculation have been observed in the wild, mainly through data-dependent hardware prefetchers [21, 22], store-to-load forwarding prediction [17], or floating point issues [19, 69], much remains to be done to fully characterize all forms of data speculation present on modern CPU platforms.

Thus, in this chapter, we ask the following questions.

Are there additional data speculation mechanisms present on modern CPUs? If so,

what are the security implications of such speculation?

4.1.1 Our Contributions

In this chapter, we answer the first question in the affirmative. More specifically, to the best of our knowledge, we document the first existence in the wild of a Load Address Predictor (LAP), where the CPU predicts addresses of load instructions based on prior addresses it had observed. In case the predicted address is cached, the CPU speculatively loads from the predicted address as opposed to the address originally intended by the program. Once the value from the predicted address is fetched, the CPU computes on it transiently using arbitrary instructions that are downstream in program code. Moreover, the speculation window is large enough for the value to be transmitted using microarchitectural side channels. Finally, we show that the LAP is present on Apple’s M2-M4 CPUs for Macs and iPads, and the closely related A15-A17 CPUs for iPhones.

Next, tackling the second question, we show the LAP is exploitable and results in grave security consequences. We first introduce Spectre-LAP, a transient 64-bit out-of-bounds read primitive that founds a new lineage of speculative execution attacks on data flows. Building on this, we introduce SLAP, an end-to-end attack on Apple’s Safari web browser capable of disclosing email content and browsing behavior from an arbitrary target webpage to a remote adversary.

Discovering and Reverse Engineering the LAP. We start by analyzing the CPU’s behavior on load instructions that cannot be reordered, due to them having a read-after-write dependency. On the M1 CPU, we observe no difference in runtime whether the load addresses increment in strides or not. However, we observe a drastic speedup on the M2 and M3 CPUs only when the load’s addresses are striding, whereas their runtime to execute the non-striding loads is similar to the M1. From this, we show that the LAP mechanism is present on recent Apple CPUs.

Next, to reverse engineer the LAP, we design a primitive that lets us observe its spec-

ulative behavior by causing a misprediction and then recovering side-channel traces of the resulting transient execution. Using this primitive, we identify that the LAP needs to be trained on 500 or more striding loads to activate reliably. When the LAP does activate, it speculatively loads from its predicted address, rather than the address dictated by the program. In turn, when the value from the predicted address arrives, we observe that the CPU opens a deep speculation window (up to 600 cycles), during which arbitrary computation can be performed on the value. This window allows us to leak the contents of LAP-predicted addresses via microarchitectural covert channels. Finally, we also observe that the stride must be at most 255 bytes for the LAP to generate predictions, limiting the scope of the LAP’s reach to this range.

Weaponizing the LAP. Now, we assume there is a secret in the address space which the adversary cannot read. While the speculation window is sufficient to leak the secret value over a cache covert channel in principle, our primitive from before would require the secret to be at most 255 bytes from the last training address in order for the LAP to transiently load it. Therefore, we modify our primitive to read from anywhere in the address space by adding another layer of indirection (i.e., an additional pointer dereference). More specifically, we write the address of the secret at a memory location reachable by the LAP. Then, we trigger an LAP prediction on this memory location, obtaining the secret’s address under speculation. We then speculatively dereference this address and leak the secret’s value using a microarchitectural covert channel. This forms the basis for Spectre-LAP. In addition, we show that Spectre-LAP can not only divert data flow, but also control flow by branching to functions which never get invoked architecturally. Finally, observing that Spectre-LAP only runs to completion on mapped addresses, we use this phenomenon to defeat Address Space Layout Randomization (ASLR) on macOS.

Orchestrating an Attack on Safari. We culminate our findings by investigating the implications of data speculation on web browser security. We port our primitive to mistrain the LAP to JavaScript, where we discover that Safari’s JavaScript engine exhibits behaviors

that are favorable for mistraining when dealing with string objects. This results in a gadget that can disclose the content of out-of-bounds JavaScript strings. However, as JavaScript's lack of pointers precludes us from using another level of indirection, the reach of our LAP gadget is again limited to read 255 bytes. We sidestep this limitation by devising a new memory massaging technique against Safari's allocator, which lands cross-origin DOM strings from the target webpage into the attacker's window. Finally, we orchestrate SLAP end-to-end, causing the LAP to disclose sensitive content from Gmail, Amazon, and Reddit when the target is authenticated.

Summary of Contributions. We contribute the following:

- We investigate the data speculation mechanisms on Apple CPUs, discovering the LAP's presence on recent generations. Next, we reverse engineer the LAP's training and activation criteria (section 4.4).
- We demonstrate that speculation using the LAP can be weaponized to achieve out-of-bounds reads anywhere in the 64-bit address space, divert control flow under speculation, and break ASLR on macOS (section 4.5).
- We build an out-of-bounds read primitive in the Safari web browser which subverts Safari's sandboxing and side-channel countermeasures, leaking cross-origin content from sensitive websites (section 4.6).

4.1.2 Responsible Disclosure

We disclosed our results to Apple on May 24, 2024. Apple's Product Security Team have acknowledged our report and proof-of-concept code, requesting an extended embargo beyond the 90-day window. At the time of writing, Apple did not share any schedule regarding mitigation plans concerning the results presented in this chapter.

4.2 Background

Cache Organization on Apple Silicon. Like most vendors, Apple CPUs use small on-chip buffers named caches to reduce the disparity in speed between the core and memory subsystem. In addition, they feature a heterogeneous core design with Performance (P) cores and Efficiency (E) cores [105]. Both types of cores have private L1 caches and shared L2 caches within a cluster of the same core type. The caches are set-associative: they are partitioned into multiple cache sets using part of the memory address, and data from that address can fit into any of the cache ways in that set, where each way contains a cache line.

Side-Channel Attacks on Caches. As the cache is shared across all processes, an adversary on the same system can measure the latency to certain data to gain inferences about the secret-dependent activity of a target process. Broadly, the plethora of prior work can be bifurcated into Flush+Reload [53, 54, 55], where the adversary times the access to shared data, and Prime+Probe [24, 25, 56, 57, 58, 59, 60, 61, 62], where they time accesses to their own data which shares a cache set with the target’s data.

Control Hazards and Speculative Execution. To improve performance, nearly all modern CPUs execute code out of program order, especially in terms of control flow. To avoid stalling when the correct control flow from a control hazard is not yet available, CPUs predict the control flow and execute instructions at the predicted execution path speculatively. If the prediction matches the control flow from program order, the changes in CPU state are committed and made visible to software. Conversely, if the prediction is incorrect, the CPU rolls back the modified state and resumes execution from the ground-truth control flow. However, microarchitectural changes, such as that to the cache, are not reverted. This leads to an abundance of control-flow speculation attacks, wherein an adversary can transiently access and recover secrets on the same system as the target [1, 2, 3, 5, 6, 11, 14, 15, 16, 31, 63, 64, 65, 66, 67, 68, 70, 71, 72, 73, 74, 75, 76].

Data Hazards and Out-of-Order Execution. In addition to predicting control flows,

modern CPUs execute instructions in the same basic block out of program order and oftentimes in parallel when their operands are made available. However, this entails three types of data hazards that must be handled: ① RAW, where the source operand of a younger instruction is the destination of an older one, ② Write After Read (WAR), where a register is read by an older instruction and modified by a younger one, and ③ Write After Write (WAW), where a register is modified twice by older and younger instructions.

Modern CPUs can resolve WAR and WAW hazards by register renaming [106], a technique that duplicates architectural registers into several microarchitectural registers to obviate the temporal dependency caused by the overwrite. In contrast, the RAW dependency is also known as a ‘true dependency’ that cannot be resolved and thus reordered. Here, the instructions must run serially due to the operand of the younger instruction being unknown.

Data Speculation on Load Instructions. However, recent industry patents and works in computer architecture propound a novel mechanism named data speculation to improve instruction-level parallelism on RAW dependencies. Here, instead of speculatively executing instructions based on predicted control flow, the CPU predicts values of data based on past execution and attempts to reorder the RAW dependency by executing younger instructions based on that speculative value [99, 100, 101, 102, 103, 104, 107, 108, 109]. Load instructions are the most common targets for data speculation since they frequently occur and highly vary in latency, thus forcing the CPU to stall for RAW hazards for more than 100 cycles on cache misses.

Load Address Prediction. One common way to speculate on load instructions is to predict the memory addresses they will access. We show an overview of a Load Address Predictor (LAP) in Figure 4.1. Here, we focus on one ARM load instruction at address `0xabcd`, which takes the data inside the `x1` register as the load address and returns the data at that address in the `x0` register.

LAPs typically keep track of the load address in `x1` each time the instruction at address `0xabcd` is executed. If the stream of addresses is predictable, such as constants or striding

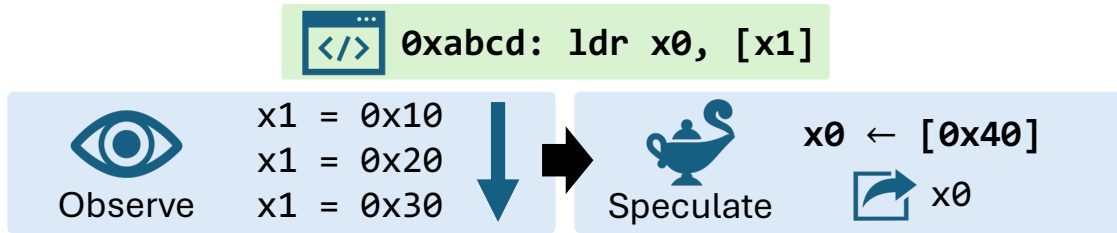


Figure 4.1: An overview of load address prediction.

values (Figure 4.1 (Left)), the LAP activates by speculatively issuing a load to the predicted address ($0x40$) and waits for it to resolve in $x0$. When the load resolves, its data can be forwarded transiently to younger instructions that use $x0$, speculating on the correctness of the LAP’s prediction. See Figure 4.1 (Right). Finally, when the load address for $x1$ resolves, speculation terminates. If the predicted address is wrong, the CPU flushes its pipeline and resumes execution from the correct load address, fetching the value to $x1$.

4.3 Threat Model

For the attack scenarios in section 4.5 and section 4.6, we target recent Apple CPUs. We assume that the target systems run macOS 14.4, which are up-to-date at the time of writing, and do not leverage software vulnerabilities. Furthermore, we assume the systems operate in their default configurations, especially with respect to side-channel countermeasures.

In section 4.5, we model the attacker as a typical native adversary with unprivileged code execution capabilities on the target system. Next, in section 4.6, we adopt the typical web-based adversary model, wherein the target visits the attacker’s webpage. Here, we assume the target uses the Safari 17.4 browser, also up-to-date at the time of writing.

4.4 Reverse Engineering the LAP

In this section, we first confirm the existence of a LAP on Apple’s M2, A15, and newer CPUs, and also rule out the possibility that our results were due to prefetching. We then reverse engineer the LAP’s activation criteria, and measure the CPU’s transient behavior

under LAP-induced speculation. Finally, we discuss limitations, such as address checks and instruction tagging.

4.4.1 Observing Load Data Speculation

We start with the following experiment outlined by the C code in Listing 4.5. Overall, we run a loop of read-after-write (RAW) dependent loads twice: first as a ‘dry run’ to bring all the load addresses into the cache to rule out the effects of classical prefetching, then second as a ‘wet run’ where we measure the runtime of the loop.

Listing 4.5: C representation of our gadget to measure the runtime of a loop of read-after-write (RAW) dependent loads.

```
1 // Dry run to cache addresses
2 volatile int array[]; // See Fig. 2
3 volatile int dep = 0;
4 for (int i = 0; i < ITERS; ++i)
5     dep = array[dep];
6 // Wet run to measure runtime
7 dep = 0;
8 uint64_t start = get_timestamp();
9 for (int i = 0; i < ITERS; ++i)
10     dep = array[dep];
11 uint64_t end = get_timestamp();
12 return end - start;
```

Line 2 starts the dry run, where we have filled an array with different contents depending on the experiment. In one experiment which we label as ‘Striding’, we fill the buffer in a pointer-chasing manner with stride S . That is, we write S to index 0, $2S$ to index S , $3S$ to index $2S$, and so on. In the other experiment labeled ‘Random’, we fill the buffer with randomly generated in-bounds indices. We illustrate this filling of the array in Figure 4.2.

Continuing the dry run in Line 3, we zero-initialize a variable `dep` to act as the RAW dependency between loads to the array. We declare the array and `dep` as `volatile` to prevent the compiler from optimizing out Lines 4-5, as `dep` is zeroed again in Line 7. Lines 4-5 show how we use `dep` to create RAW-dependent loads in a manner similar to traversing a singly-linked list. The load address into the array for an arbitrary iteration i cannot be

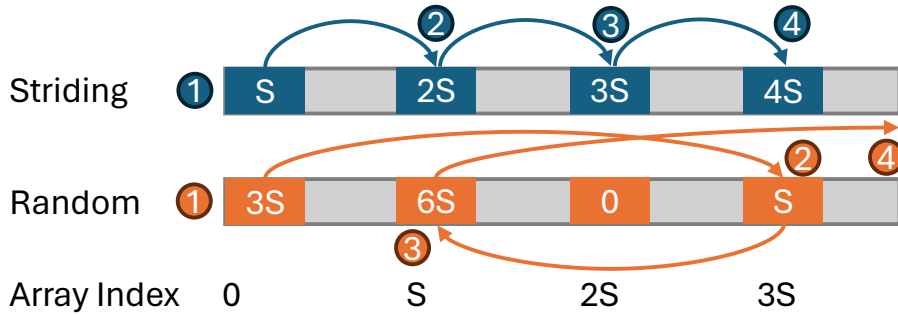


Figure 4.2: The contents of the array in Listing 4.5 for each experiment. The numbered bulletpoints indicate the order of memory accesses, with the first memory access starting at the beginning of the array.

determined until the load of iteration $i - 1$ resolves, and its load value is copied into `dep`. As a RAW dependency is a true dependency that cannot be resolved by register renaming (cf. section 4.2), we expect the loads to be serialized on typical CPU microarchitectures, regardless of the load addresses or values.

We now move on to the wet run, where Line 8 obtains a timestamp. Lines 9-10 are identical to Lines 4-5 during the dry run, looping through the RAW-dependent loads. Finally, Line 11 obtains a second timestamp, and the difference from the first timestamp is returned in Line 12.

Experimental Setup. We run Listing 4.5 on the P- and E-cores of the Apple M1, M2, and M3 CPUs, using the `pthread_set_qos_class_self_np` API in macOS to schedule the Constant and Random experiments on each core type. For both experiments, we report the median runtime from 100 invocations of Listing 4.5. We set $S = 32$ bytes, and increase the `ITERS` variable (in Lines 4 and 9) from 10 to 1,000 in increments of 10 iterations. Given these parameters, the maximum size for the array is $1000 * S < 32$ KiB. The L1 data cache size of the M-series CPUs is 128 KiB for the P-cores and 64 KiB for the E-cores [105], and thus all array elements can remain cached. Finally, for the `get_timestamp` calls in Lines 8 and 11, we use macOS’s `kperf` API to count CPU cycles.

Results. We show the resulting plots in Figure 4.3. On both core types of the M1 (top

left and bottom left), we observe a consistent linear increase in latency, which does not differ based on the load addresses or values. On the E-cores of the M2 (bottom center), we observe an identical trend. Thus, we conclude these cores lack load data speculation mechanisms.

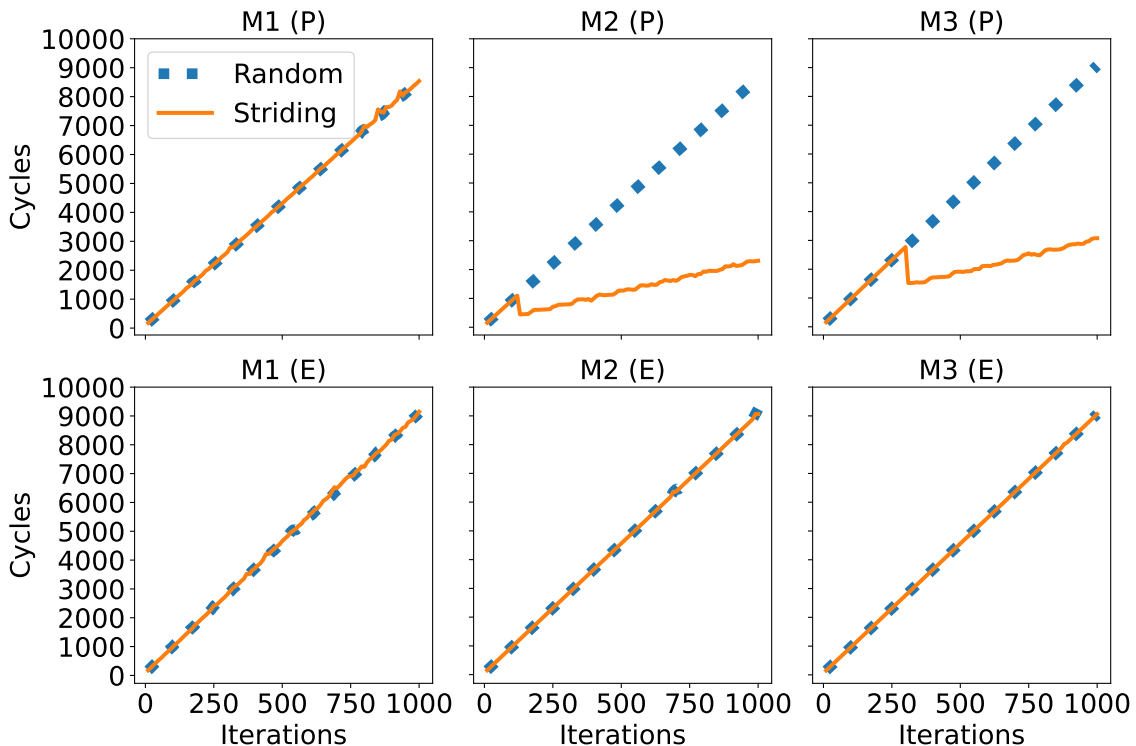


Figure 4.3: Striding (solid) and Random (dotted) experiment plots for the P- and E-cores of the Apple M1, M2, and M3 CPUs.

However, on the P-cores of the M2 (top center), the plot is vastly different: while the latency for the Random experiment continues to increase linearly, the latency for the Striding experiment diverges around 120 iterations, where we observe a notable speedup. As the number of iterations increases, the latency for the Striding experiment grows much slower compared to that of the Random experiment. Given that this speedup occurs even though the loads are RAW-dependent on each other and cannot be parallelized, and occurs only on loads whose addresses and values exhibit a pattern, we confirm the presence of a data speculation mechanism for loads on the P-cores of the M2.

Next, we attribute the obtained speedups to instruction-level parallelism. That is, with-

out load data speculation, younger RAW-dependent loads must stall until older loads are resolved. Conversely, a CPU with data speculation can transiently execute the loads (using predictions) in parallel with verifying these predictions from resolved loads.

Finally, we observe similar behavior on the M3, with the E-cores showing no notable difference between the Striding and Random experiments (bottom right), but the P-cores showing a speedup on the former (top right). Moreover, whereas 120 striding loads suffice to activate the prediction mechanism on the M2, we observe this threshold is higher on the M3 where the speedup occurs at 320 iterations.

4.4.2 Confirming Load Address Prediction

Having observed speedups from load data speculation, we seek to confirm that the mechanism is a LAP. To test for one, we fix the load values to be random. Then, we introduce the Striding Addresses from Random Values (SA+RV) experiment where the load addresses are striding nonetheless. The top half of Figure 4.4 shows the array contents and memory access pattern that we desire.

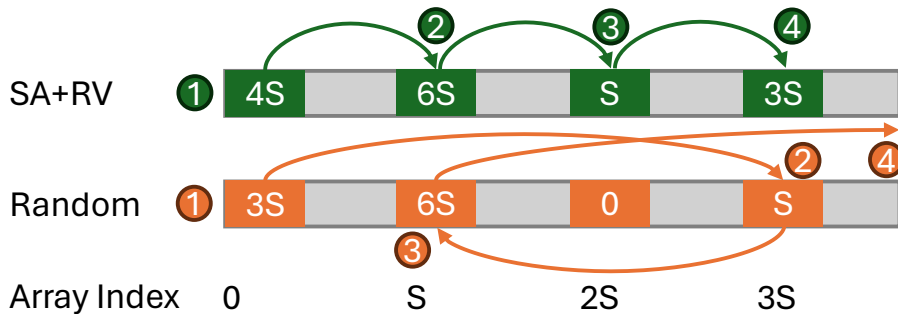


Figure 4.4: Memory contents and access pattern for the SA+RV and Random experiments. The latter is identical to the bottom half of Figure 4.2.

Despite the load values being random multiples of S from our array fill, the array accesses always stride by S in the SA+RV experiment. We compare it to the Random experiment from subsection 4.4.1 at the bottom half of Figure 4.4, where both the load addresses and values are random. That is, we confirm the existence of a LAP if we observe a speedup on the SA+RV experiment compared to the Random experiment. To achieve the memory

access pattern of the SA+RV experiment, we modify the for-loop of Listing 4.5 (Lines 3-4 and 9-10) slightly, as shown in Listing 4.6.

Listing 4.6: Modified for-loop for making load addresses stride, despite the contents of the array being random.

```

1  for (int i = 0; i < ITERS; ++i)
2      dep += min(array[dep], S);

```

Instead of assigning the load value directly to the `dep` variable to be used as a RAW dependency for the next load address, we increment `dep` by the smaller of the load value and the stride S . Furthermore, we populate the array with randomly generated nonzero multiples of S , such that regardless of the load values the array accesses will be `array[0]`, `array[S]`, `array[2S]`, and so on. We run the Random and SA+RV experiments on the P-cores of the M2 and M3 CPUs. For the SA+RV experiment, we use the same parameters (median of 100, $S = 32B$, and cycle-counting) as the Random experiment. Figure 4.5 shows the resulting latency plots.

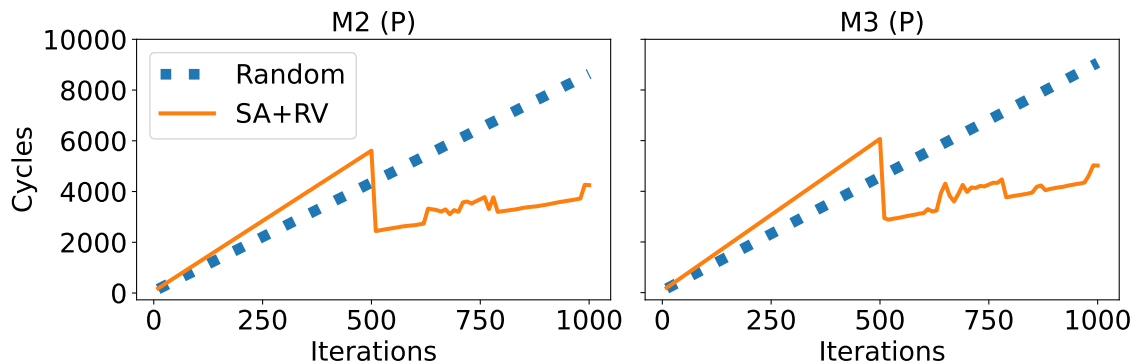


Figure 4.5: SA+RV (solid) and Random (dotted) experiment plots for the P-cores on the Apple M2 and M3 CPUs.

On both the M2 (left) and M3 (right) P-cores, until 500 iterations, the latency for the SA+RV experiment grows faster than the Random experiment due to the additional overhead of the `min` function in Line 2 of Listing 4.6. At 500 iterations and beyond, though, the SA+RV latency sharply drops below that of Random and stays below. Moreover, it grows

at a notably slower rate.

From this, we arrive at an important conclusion: the M2 and M3 CPUs observe load addresses originating from the same instruction address, predicting subsequent load addresses when the previous ones are striding. Thus, we determine that their data speculation mechanism is a LAP.

Benchmarking More Apple Silicon. Next, we extend our experiments to Apple’s recent mobile CPUs (A-series), which share core designs with the M-series [105]. However, A-series CPUs are available only on iOS devices, where we do not have access to CPU cycle counters. As such, we develop a portable version of the Random, Striding, and SA+RV experiments by compiling them to WebAssembly. We then compile the WebAssembly binary to machine code using WebKit, the engine underlying the Safari web browser. Since the timer resolution in WebKit is coarse (1 ms), we fix the number of iterations to one million to amplify the timing difference, while keeping the stride unchanged at 32 bytes. We indicate the presence or absence of the LAP across several devices in Table 4.1.

Table 4.1: LAP presence/absence on recent Apple devices.

Device	CPU	LAP Present?
MacBook Pro (A2338)	M1	No
MacBook Air (A2681)	M2	Yes
MacBook Pro (A2918)	M3	Yes
iPhone 11 (A2111)	A13 Bionic	No
iPhone 12 (A2172)	A14 Bionic	No
iPhone 13 Mini (A2481)	A15 Bionic	Yes
iPhone 13 (A2482)	A15 Bionic	No
iPhone 14 Pro Max (A2651)	A16 Bionic	Yes
iPhone 15 Pro Max (A2849)	A17 Pro	Yes
iPad Pro (7th Gen.) (A2925)	M4	Yes

We note that the core design of the M1 is similar to the A14 Bionic, M2 to A15 Bionic, and M3 to A16 Bionic [105]. Thus, we observe results on the A-series devices that are mostly consistent with our previous observations on M-series devices. However, we observe contrasting results on two devices that were released simultaneously (iPhone 13 Mini

and iPhone 13) and have the same A15 Bionic CPU. Given that the A15 is manufactured by only one firm (TSMC), to the best of our knowledge, this is the first observance of Apple CPUs with the same product code differing in microarchitectural behavior in a similar manner to the stepping level in Intel and AMD CPUs. Finally, we note that the M4 CPU was just released at the time of writing and also contains the LAP like its predecessors.

4.4.3 Confirming Speculative Execution via LAP

We now recall from section 4.2 that LAPs speculatively execute the load on the predicted address, bring the load’s value into the register file, and then use the value for younger dependent instructions until the ground-truth load address becomes known. While the drastic speedups from the Striding and SA+RV experiments in subsection 4.4.1 and subsection 4.4.2 strongly point at such behavior, the loop of RAW-dependent loads causes difficulties for reverse engineering because in the case the LAP activates on striding addresses, the speculative execution is always correct. This causes it (and its microarchitectural traces) to be ‘masked’ by architectural execution, precluding us from precisely measuring behavior within the LAP’s speculation window.

Therefore, in this subsection, we aim to collect stronger evidence for LAP-induced speculation and lay the groundwork for subsequent reverse engineering experiments by causing the predicted load address to differ from the ground-truth load address, resulting in misspeculation that leaves different microarchitectural traces.

Gadget Setup. We first implement a singly-linked list where each node has a pointer to the next node and a pointer to data in Figure 4.6. Here, we aim to mistrain the LAP using the loads to the data elements.

As shown in all the nodes but the last, the nodes’ data pointers hold striding offsets of a writable buffer in memory. In each of the offsets, we write a dummy value. However, we break this striding behavior for the last node. That is, to the next striding offset of the buffer, we write a secret value this time. But the last node’s data pointer does not point

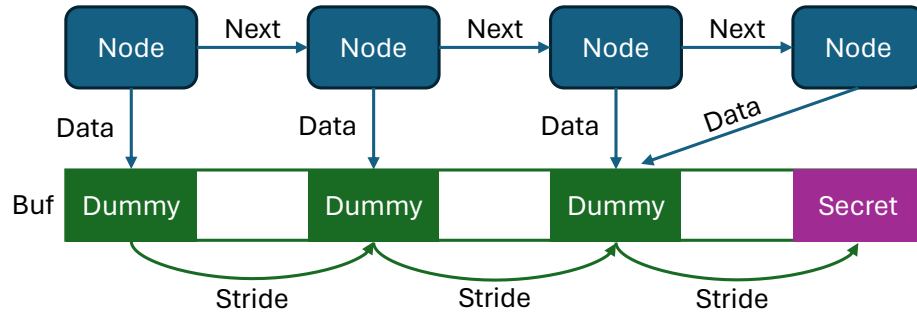


Figure 4.6: A graphical overview of the linked list and buffer data structures to cause misprediction by the LAP. These data structures are also used by the code shown in Listing 4.7.

there, instead pointing to the same memory address as the second-to-last node. As such, all architectural dereferences to data only point to dummy values.

Inducing Misprediction. Next, using the C code in Listing 4.7, we show how to train the LAP such that it predicts a load address that is never architecturally accessed and transmits its value over a covert channel.

Listing 4.7: C representation of our misprediction gadget. The gadget performs a linked list traversal while dereferencing the data pointer of each node and encoding the value into a cache channel.

```

1  cache_flush(last_node, FR_buf);
2  struct Node *n = first_node;
3  while (n != NULL) {
4      uint8_t LAP_load = *(n->data);
5      FR_buf[LAP_load * PAGE_SZ];
6      n = n->next;
7  }
8  return FR_recv(FR_buf);

```

We use Flush+Reload [54] as our covert channel for experimental expediency, as it supports transmitting several different byte values. Line 1 performs its flushing step, but also flushes the last linked list node from the cache. By doing so, the value of the last node’s data pointer is not quickly retrievable, forcing the CPU to use the predicted address from the LAP. We show the effects of running Listing 4.7 on Figure 4.6’s data structures in Figure 4.7. First, we illustrate Line 1’s effect at the top right corner, with the last node

being colored orange to indicate it is uncached.

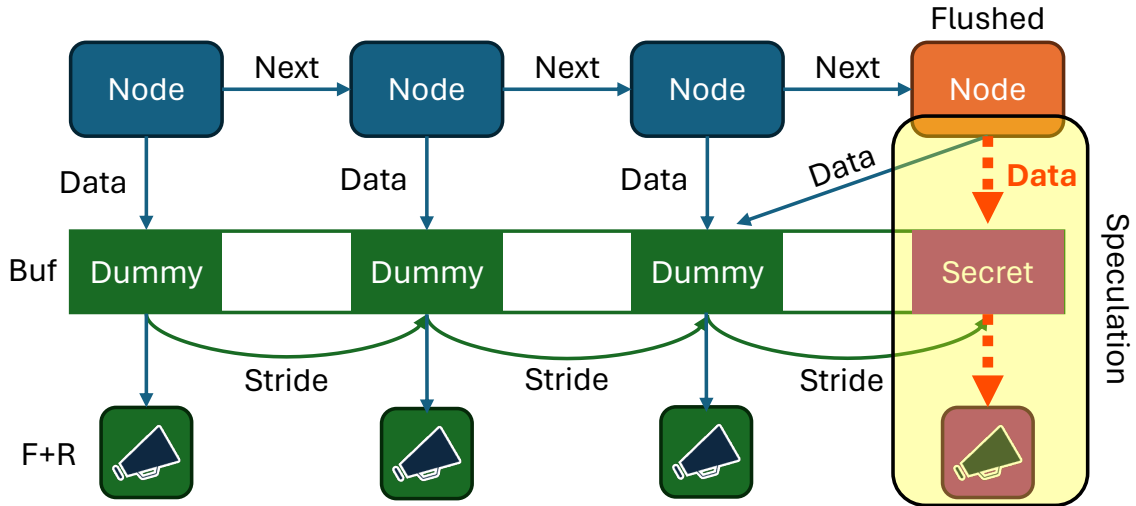


Figure 4.7: Overview of the outcome of running Listing 4.7 on Figure 4.6. There is a divergence between architectural execution (blue arrows) and speculative execution (red arrows, highlighted area).

Next, Line 2 declares a pointer to the first linked list node, and Line 3 starts a while-loop for the traversal. In this loop, Line 4 dereferences the data pointer of the current node, storing its value in the `LAP_load` variable. It is this load instruction which we coerce the LAP to predict the address of, as for all nodes but the last, the load addresses become B , $B + S$, $B + 2S$ and so on for a given stride S and buffer address B . Hence, at the last node, the LAP's predicted load address becomes $B + nS$ for a linked list of length $n + 1$, whereas the actual load address is $B + (n - 1)S$. The right side of Figure 4.7 depicts this, where the speculative dereferences (red arrows) differ from architectural, putting the secret value into `LAP_load`.

Finally, Line 5 transmits the load value, and Line 6 advances to the next linked list node. We then receive over the covert channel on Line 8. Without any LAP activation, we expect to receive the dummy value only. On the other hand, with LAP activation (and misprediction), we expect to receive both the secret and dummy values.

This is shown at the bottom of Figure 4.7, where the bullhorn icons represent covert channel transmissions. The area highlighted in yellow marks the speculative execution

from the mispredicted load address, which leads to the secret value being transmitted.

Experimental Setup for Reverse Engineering. We now introduce the setup we use to collect data from the workload in Listing 4.7, and for further reverse engineering experiments in subsection 4.4.4, subsection 4.4.5, and subsection 4.4.6. We focus on the LAP of the Apple M2 CPU. For accurate measurements, we require the ability to count cycles and flush cache lines from userspace programs. Furthermore, to reduce noise and variation, we require the ability to manually control CPU frequency, isolate CPU cores from being used by operating system processes, and pin programs to the isolated cores.

To that aim, we perform our experiments on Fedora Linux’s Asahi Remix with kernel version 6.6.3-414, since macOS lacks support for all of our requirements except cycle counting. Here, we set all cores to their maximum frequency using the `cpufreq` interface in `sysfs` and the userspace governor. We exclude one P-core using the `isolcpus` kernel parameter, and pin our experiment to it with the `sched_setaffinity` system call.

Initial Results for Measuring Speculation. Following prior results on Figure 4.5 where we observed LAP activation at 500 iterations, we use a linked list of 501 nodes to train the LAP on 500 striding data pointer loads, which are made RAW-dependent by the linked list traversal. Likewise, we keep the stride between data pointer addresses at 32 bytes. Next, we test our gadget 1,000 times, where in each trial we repeat Listing 4.7 for 20 runs. Over the Flush+Reload covert channel, we receive the dummy value on all 1,000 runs and the secret value on 721 runs. We show a histogram of the latency to reload the secret value in Figure 4.8.

Finally, we ascertain our results by not writing the secret value into the striding buffer offset, and confirming that it is no longer received over the covert channel. Hence, we demonstrate that we can dereference pointers under speculation and make the CPU operate on data at rest by the LAP. That is, after being written, the secret value is never architecturally read in by the core.

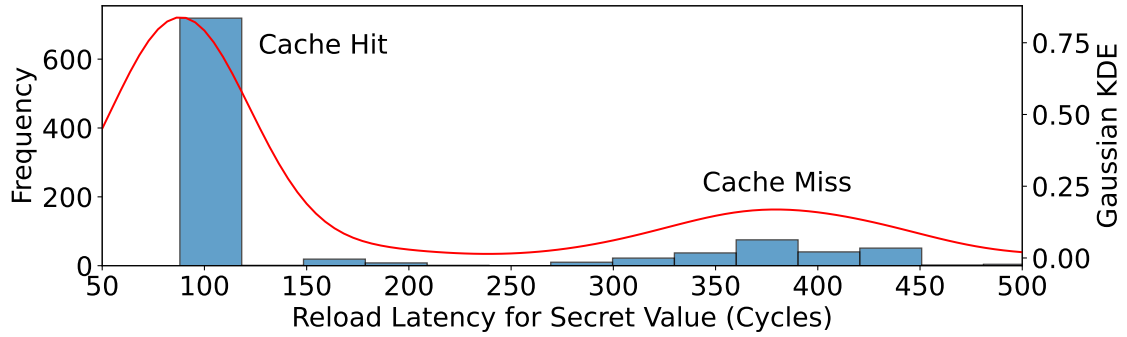


Figure 4.8: Histogram of latencies to the Flush+Reload memory address that corresponds to the secret value.

4.4.4 Spatial Conditions for Speculation

Having observed LAP-induced speculation from a mispredicted load address, we now seek to identify the spatial conditions to reliably activate the LAP, that is, in terms of virtual addresses the LAP interacts with and their topology. We observe that the training length (i.e., number of linked list nodes) and the stride for the load addresses are both parameters, and analyze how they affect the probability of the LAP activating. Then, we ask questions about the paging of addresses and their effect on LAP training or activation, as most classical prefetching occurs within a page.

Training Length and Stride: Experimental Setup. Our setup here is largely identical to the experiment in subsection 4.4.3, where we run our misprediction gadget 1,000 times on the same set of parameters. However, we vary the linked list length from 20 to 1,000 in increments of 20, as well as the stride between pointers to the buffer from -320 bytes to 320 bytes in increments of 8 bytes. That is, for a negative stride value, we start writing the first dummy value at the end of the buffer, and traverse it backwards with the given stride to train the LAP.

Training Length and Stride: Results. We plot our results in Figure 4.9 as a heatmap. Here, training length and load address stride are the X- and Y-axes, and we represent the number of observed LAP activations (out of 1,000) as the heat, where the closer the color

to dark blue, the more the LAP activates. Also, we highlight zero values in yellow.

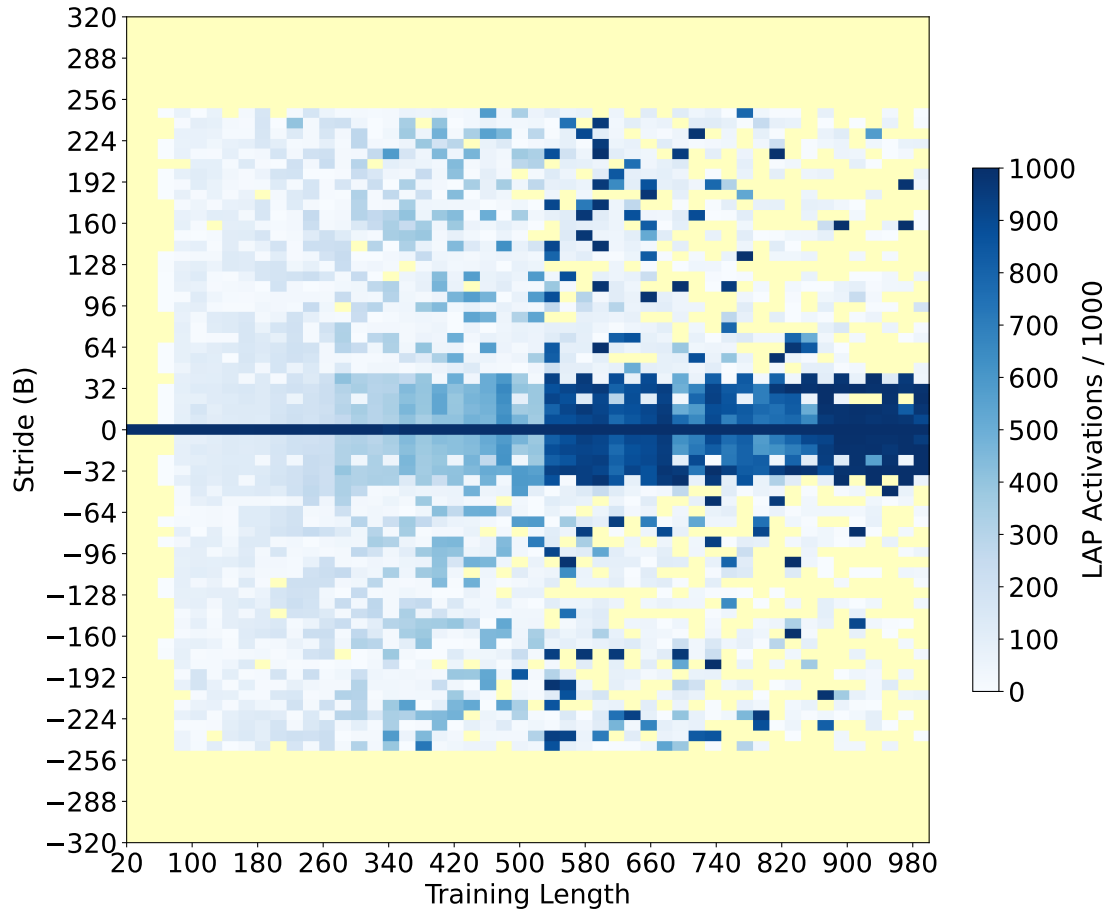


Figure 4.9: Heatmap showing the effects of the linked list length and the stride between training load addresses on the likelihood of LAP activation. Values of zero are highlighted in yellow.

Firstly, at Stride = 0, the LAP seems to always activate regardless of training length. However, this is expected, as a zero stride leads the data pointers of all nodes to point to the start of the buffer. In turn, this causes the secret value to be transmitted architecturally. Hence, this is not speculation, and we disregard the result (horizontal blue line).

Dead Zones for LAP Activation. Secondly, we observe three ‘dead zones’ in Figure 4.9 with no LAP activations (highlighted in yellow). The first is to the left side of the heatmap, when the training length is less than 80. As we can now measure LAP activity much more precisely by checking for misprediction, we regard this as the minimum training threshold

for RAW-dependent striding load addresses from which we can observe LAP activity (albeit at a low rate). Furthermore, the second and third dead zones are at the top and bottom of the plot, where the absolute value of the stride is 256 bytes and above. From this, we confirm that the LAP can keep track of both positive and negatively striding load addresses, and conjecture that its internal state for the stride is one sign bit with eight ($\log_2 256$) magnitude bits.

Optimal Conditions for LAP Activation. Thirdly, we observe regions of both training length and stride wherein the LAP has a proclivity to activate. We note a sharp increase in activations across most strides with magnitude less than 256 bytes when the training length exceeds 500. This threshold coincides with when we observed an abrupt speedup while testing for a LAP in subsection 4.4.2, hence we regard this as the practical training threshold to regularly observe LAP activations. On the other hand, across all training lengths above 80 (and especially past 500), we emphasize a ‘block’ of frequent activations when the stride has a magnitude below 64 bytes. Coincidentally, this is the size of an L1 cache line on Apple CPUs: we conjecture that small strides that access a cache line more than once during training increase the LAP’s inclination to activate.

Measuring LAP Training and Prediction Across Page Boundaries. We also demonstrate that the LAP on the M2 maintains training state across page boundaries, but will not generate a prediction on a new page. See Appendix A.

4.4.5 Temporal Conditions for Speculation

Moving away from spatial conditions, we focus on identifying the temporal conditions for starting LAP-induced speculative execution and prolonging the speculation window. First, we measure the window for which the LAP keeps state during training when given extraneous instructions. Then, we measure how deeply the LAP speculates conditioned on the caching status of the variables that are crucial to the linked list traversal.

Gadget Overview. We modify the loop of Listing 4.7 (Lines 3-7) into Listing 4.8 to better

separate (at the source code level) the training iterations from the loop iteration where the LAP misprediction happens.

Listing 4.8: Rewritten linked list traversal loop from Listing 4.7 that allows us to insert dummy instructions at two different points in the code to measure the training and speculation windows.

```
1  uint8_t LAP_load;
2  for (int i = 0; i < LL_SIZE; ++i) {
3      // Insert MULs for training window.
4      LAP_load = *(n->data);
5      n = n->next;
6  }
7  // Insert MULs for speculation window.
8  FR_buf[LAP_load * PAGE_SZ];
```

This loop is functionally equivalent for traversing the linked list and dereferencing each node's data pointer, thereby training the LAP. However, on Line 2, we use a for-loop spanning the linked list's length instead of a while-loop to make the loop unrollable by the compiler. Finally, we move the Flush+Reload transmission outside the loop to Line 8, having declared the `LAP_load` variable in Line 1 to have it in scope. As the loop terminates immediately after LAP activation on the last node, the covert channel reception is identical to before: LAP activation results in the secret and dummy values being transmitted, while no activation transmits just the dummy value.

However, the most notable difference is that we introduce two lines where we can add extraneous `mul` instructions. If we insert them in Line 3, they are architecturally executed between each load that trains the LAP in Line 4. This lets us measure the effect of extraneous instructions during LAP training. On the other hand, if we insert them in Line 7, they are also speculatively executed before the covert channel transmission when the LAP activates. Hence, here we can measure the speculation window's length.

Measuring the Training Window. We run our misprediction gadget for 1,000 runs on a stride of 32 bytes and 1,000 training addresses, since we have identified these parameters to be optimal for activating the LAP from subsection 4.4.4. We continue using these pa-

rameters for all experiments in this subsection. We use Listing 4.8 for the traversal loop, and insert `mul` instructions into Line 3 from zero to ten of them in increments of one. We plot the effect of the extraneous `mul` instructions on the number of LAP activations in Figure 4.10.

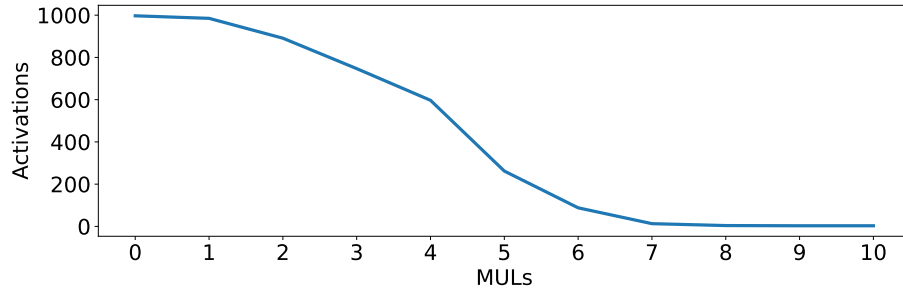


Figure 4.10: Number of LAP activations out of 1,000 runs when extraneous `mul` instructions are inserted during training.

Surprisingly, we observe a sharp decrease in activations past 2 `mul` instructions, to no activations at 8 `mul`s and more. Hence, we conclude that the LAP employs a short training window, wherein the striding loads must be executed quickly enough to train the LAP. With Apple documentation stating that each `mul` requires 3 cycles [105, Appendix A], this window is at most 24 cycles.

Measuring the Speculation Window. We repeat the setup for measuring the training window. Rather than inserting the `mul` instructions into Line 3 of Listing 4.8, we insert them into Line 7 instead (such that they will execute speculatively) from 0 to 300 of them in increments of 10.

We measure the speculation window conditioned on the caching status of the last linked list node and the predicted address. Firstly, we keep the predicted address cached, and measure how many `mul` instructions can fit when the last node is cached or flushed in Figure 4.11 (Left). Then, we flush the predicted address and repeat the experiments in Figure 4.11 (Right).

When the predicted address is cached, we observe that the window length when the last node is cached is similar to that of a speculation window opened by a branch predictor

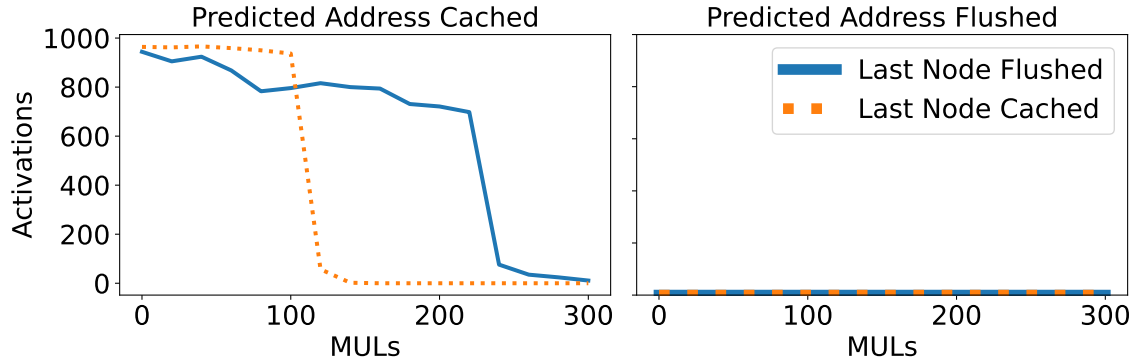


Figure 4.11: Number of LAP activations when `mul` instructions are placed in the speculation window. (Left) The LAP’s predicted address is cached. (Right) The predicted address is flushed.

when the predicate is evicted, at around 100 `mul`s [98, Section 4.2]. Contrarily, when the last node is flushed, we show a speculation window twice as deep, exceeding 200 `mul`s (or 600 cycles). However, with the experiments where the predicted address is flushed, we observe 0 activations regardless of the caching status of the last node. Hence, we conclude that another precondition for the LAP to activate is for the predicted address to be cached, and that the LAP does not prefetch its predicted address.

Comparison with Data-Dependent Prefetching. Now that we have understood how the LAP behaves, we revisit the Data Memory-dependent Prefetcher (DMP) that was shown in prior work [21, 22] to be present also on recent Apple CPUs. If data being returned from a load resembles a valid pointer, the DMP treats the data as an address and dereferences it, anticipating that the address will be loaded from at some later point. Although the LAP and DMP are both present on Apple CPUs and activate on load instructions, we note some important distinctions. Unlike the LAP, the DMP does not speculate past RAW dependencies, since it is a hardware prefetcher. Conversely, the LAP is not a prefetcher: from the speculation window experiments in this subsection, we observed that the LAP terminates if the predicted address is not cached, instead of prefetching it and continuing. Finally, only the LAP opens a speculation window wherein arbitrary computations can be performed.

4.4.6 Confirming Instruction Address Tagging

We recall that typical LAPs proposed in literature keep state per instruction address (cf. section 4.2). Using the experiments in Appendix B, we first observe that unrolling Listing 4.8 causes the LAP not to activate, confirming that it uses the instruction address as a tag. Subsequently, we test if the LAP uses all or part of the instruction address for such tagging. We begin by observing that the LAP’s training state persists even if training is interrupted at the 30th-to-last node. Next, we traverse the last 30 nodes using a clone of Listing 4.8 whose instruction address aliases the original for the lowest 6 to 47 bits. Finally, we confirm that the LAP uses all canonical address bits of the training loads as a tag.

4.5 Weaponizing the LAP for Attacks

Having confirmed the LAP’s existence and described techniques to use it for speculative execution in section 4.4, we now weaponize these techniques to serve as proof-of-concepts on the danger of LAPs. We show how the linked list traversal which activates the LAP can be modified to speculatively hijack both data and control flow, just with one more load in the traversal pattern. We then use this new technique, which we name Spectre-LAP, to break Address Space Layout Randomization (ASLR) on macOS.

4.5.1 Spectre-LAP

With our misprediction experiment in subsection 4.4.3, we recall from Figure 4.7 that with a load address stride of S , the LAP gets trained to jump S bytes beyond the last dummy value in the buffer. However, from an adversarial perspective, the out-of-bounds reach becomes limited to S which we show in subsection 4.4.4 is at most 255 bytes. As such, we now aim to augment our primitive with the LAP’s deep speculation window shown in subsection 4.4.5 to transiently reach anywhere in the address space.

Gadget Overview. We revise the experiment in subsection 4.4.3 to use the data structures

shown in Figure 4.12. We assume there is a secret at address `addr` that we wish to read. For this, we retain the linked list with data pointers with striding data pointers, and we flush the last node to force the CPU to use the LAP's predicted address as before.

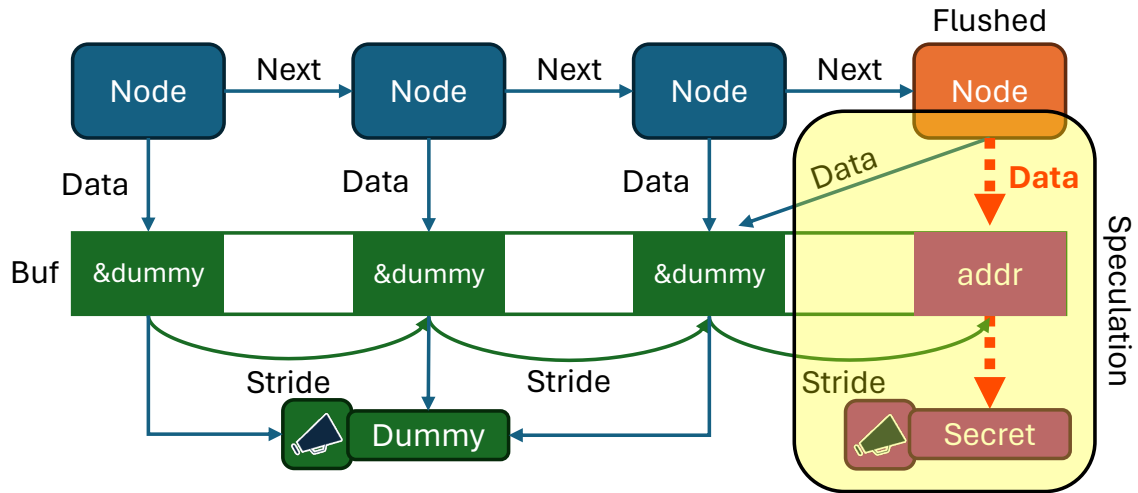


Figure 4.12: Our modified misprediction primitive that speculatively dereferences double pointers, allowing for a 64-bit out-of-bounds read. The bullhorns indicate that both the dummy and secret data will be transmitted over the covert channel.

However, our goal is to hijack data flow further beyond S bytes of the last dummy in the buffer. To that aim, we add a level of indirection by treating the contents of the buffer as addresses, as opposed to values. We declare each node's data pointer to be a double-pointer, and write the dummy element's memory address at the buffer's striding offsets. Next, we write `addr` at the last striding offset (i.e., the LAP's predicted address), but make the last node's data pointer point to the last dummy address written into the buffer. In turn, for every node we traverse, we dereference the double-pointer twice and transmit the resulting value over the covert channel, which we depict with bullhorns. Hence, LAP misprediction on this linked list results in the LAP first loading `addr`, then loading the secret, then finally transmitting the secret as we show in Figure 4.12's highlighted box. Lastly, we show the traversal code corresponding to our new linked list in Listing 4.9.

The sole change we make is Line 2 (highlighted). Here, the first dereference into the buffer is the speculative load by the LAP, and retrieves `addr`. When the LAP speculatively

Listing 4.9: C representation of the linked list traversal code for Figure 4.12, i.e., a gadget for Spectre-LAP.

```
1 while (n != NULL) {  
2     uint8_t LAP_load = *(n->data);  
3     transmit(LAP_load);  
4     n = n->next;  
5 }
```

forwards `addr`, the second dereference loads the secret into `LAP_load`, which gets leaked in Line 3.

4.5.2 Hijacking Control Flow

Now, we show a variant of Spectre-LAP that diverts control flow under speculation in addition to data flow. Previously in subsection 4.5.1, the dummy and secret elements in Figure 4.12 were buffers containing data. Now, we assume there is a function we wish to call during speculation named `secret_func`, whose entry point is at address `f_addr`. To divert control flow, we replace the dummy buffer with a dummy function that shares a signature (arguments and return type) with `secret_func`. In turn, we write the address of the functions to the buffer, making each linked list node contain a double function pointer. Graphically, we keep the linked list structure at the top half of Figure 4.12, but replace the contents of the buffer and the memory pages for the secret and data elements with Figure 4.13.

Invoking Rogue Functions Under Speculation. In this modified setup, the dummy function does nothing and returns. Conversely, `secret_func` contains the covert channel transmission, followed by a load to an invalid address. This ensures that our program will segfault if the secret function is ever invoked architecturally. Next, we replace Listing 4.9 with Listing 4.10 to handle function pointers.

Comparing Line 2 (highlighted) with that of Listing 4.9, the first dereference is still the load which the LAP activates on. However, at that point, the second dereference now

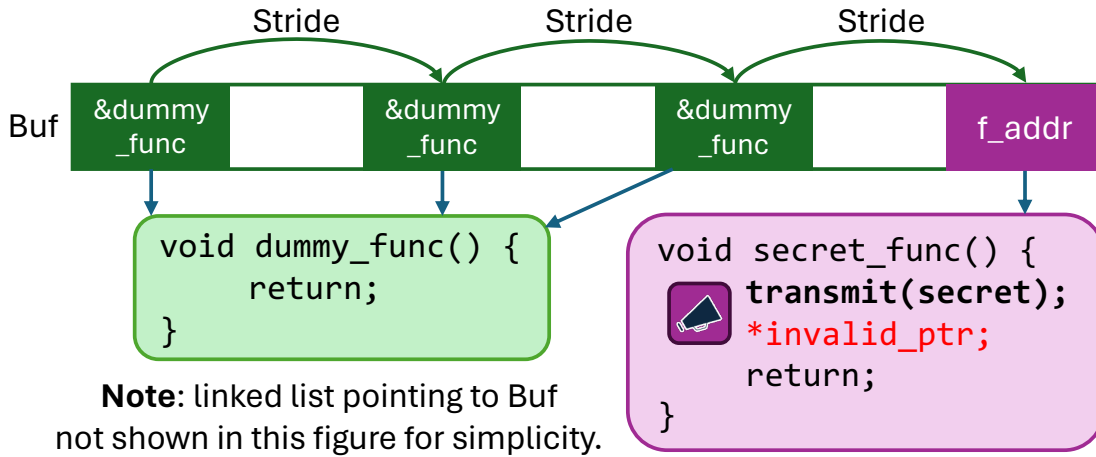


Figure 4.13: Our changes to the lower half of Figure 4.12 for the pointer values in the buffer to point to dummy and secret functions.

Listing 4.10: C representation of the modified Spectre-LAP gadget to dereference function pointers.

```

1 while (n != NULL) {
2     *(n->data) (); // Call pointed function
3     n = n->next;
4 }

```

brings the entry point of the secret function (`f_addr`) into the CPU’s register file. As the CPU continues to speculate, it branches to the entry point and executes the function body, which contains the covert channel transmission (instead of within Listing 4.10). As such, if the LAP activates and diverts control flow under speculation, we will receive a value over the covert channel. In contrast, if the LAP fails to speculatively call functions, we will not receive any value.

4.5.3 Evaluation and Breaking ASLR

We now evaluate the accuracy and throughput for both variants of Spectre-LAP on the M2 and M3 CPUs. Moreover, when using our attack on an invalid address, we find that speculation continues to the covert channel transmission only when we dereference mapped memory under speculation. This serves as our motivation for breaking ASLR.

Attack Setup. For all three attacks in this section, we report the median value of 100 runs. Moreover, as macOS disables cache flush instructions, we use Evict+Reload as the covert channel instead. We continue to use the optimal LAP training parameters of 1,000 addresses striding 32 bytes apart from subsection 4.4.5 and subsection 4.4.6.

Out-of-Bounds Reads. We first measure the out-of-bounds read primitive from subsection 4.5.1 by reading a secret string. Here, we report the leak rate and bitwise accuracy when recovering the string. The top row of Table 4.2 shows our results, where we demonstrate that we can read out of bounds robustly on both CPUs. Though, we observe the LAP is more difficult to activate on the M3, indicated by its significantly lower throughput.

Table 4.2: Results for Spectre-LAP and ASLR break attacks on the M2 and M3 CPUs, shown as the median of 100 trials. The baseline accuracy, i.e., randomly guessing the ASLR slide, is $1/5120 \approx 0.0002$.

Attack	M2 Acc.	M2 Rate	M3 Acc.	M3 Rate
OOB Reads	1.00	9,140 b/s	1.00	5,795 b/s
Control Flow	0.99	9,481 runs/s	0	0 runs/s
ASLR Break	0.72	11.39 ms	0.44	1,299.26 ms

Hijacking Control Flow. Then, we measure the control flow hijacking primitive from subsection 4.5.2. In this case, we measure throughput as how quickly we can execute the misprediction routine, and accuracy as the fraction of executions that resulted in a covert channel transmission. In the middle row of Table 4.2, we observe similar results as reading out-of-bounds on the M2. However, we do not receive anything over the covert channel on the M3. Thus, we conjecture that newer Apple CPUs contain measures that prevent function calls from executing under speculation.

Breaking ASLR on macOS. XNU (the kernel underlying macOS) employs a maximum slide of 80 MiB, placing the entry point of a Mach-O binary at the start of any 16KiB page between virtual address $0 \times 100,000,000$ and $0 \times 105,000,000$ [110, 111, 112]. The ‘magic’ byte string denoting the start of every 64-bit Mach-O binary is $0 \times \text{feedfacf}$ [113], hence we repeatedly search for this sequence using Listing 4.9 amidst

5,120 possible pages. We measure the time to find a match, and check our guess against the ground-truth slide value using the `vmmap` utility.

We succeed in breaking ASLR on both the M2 and M3, and show the results in the bottom row of Table 4.2. We find the M2’s more amenable LAP beneficial, recovering the correct slide on 72 runs in a median time of less than 12 ms. In contrast, on the M3 the search must be repeated several times until we can output a guess, slowing the runtime by two orders of magnitude and also affecting accuracy.

4.6 Attacking Safari With the LAP

We now reason about reading out-of-bounds in the JavaScript sandbox of the WebKit engine underlying the Safari web browser. Firstly, we describe how to overcome the restrictions set by JavaScript sandboxing such as the lack of pointers, memory management, and high-resolution timing, resulting in a gadget that retains the linked list structure but is able to transiently read JavaScript string objects which do not belong to the attacker’s webpage. Secondly, we further discuss and leverage WebKit’s memory allocation techniques, which enable us to place an arbitrary JavaScript string from any target webpage within the out-of-bounds reach of our browser-based gadget.

These two parts complete our attack, which we name SLAP, a portmanteau of speculative execution on data, Safari, and the LAP. Finally, we show how SLAP can compromise the security of the Safari web browser, reading login-protected data from the DOM of real websites.

4.6.1 Timer-Resilient Covert Channel

As a side-channel countermeasure, WebKit restricts the timer resolution in JavaScript to 1 ms [48, 98]. Hence, for an end-to-end attack, we must first be able to distinguish cache hits from misses with extremely coarse timing, about seven orders of magnitude coarser than the timing difference between a single cache hit and miss. We achieve this with an

amplifier gadget which we detail in Appendix C.

4.6.2 SLAP Gadget Overview

We recall that the out-of-bounds read primitive in Listing 4.9 uses two data structures: a linked list with data attached, and a buffer that has pointer values written into it at a fixed stride. The former is possible in JavaScript, but pointers do not exist in the language. After observing that the metadata object of JavaScript strings in WebKit is 16 bytes wide, and that hundreds of them can be contiguously allocated, we replace the buffer with a region of WebKit’s heap for JavaScript string objects full of dummy strings.

Training the LAP on String Objects. We illustrate the changes for training the LAP on JavaScript strings in Figure 4.14. Similarly to the native version in subsection 4.4.3, we adopt a construction where the last linked list node is evicted to prolong the LAP’s speculation window (cf. subsection 4.4.5), and where each node’s data reference is a training address.

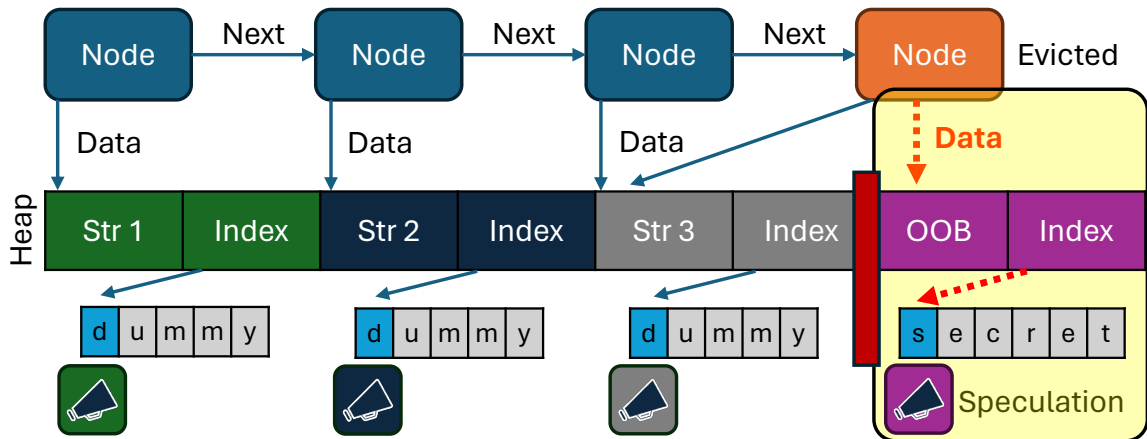


Figure 4.14: Graphical overview of the browser-based version of our LAP-training gadget. Architectural execution is shown in blue arrows, while speculative execution is shown in red arrows and the highlighted region.

In this case, the data are JavaScript strings owned by the attacker’s webpage, labeled Str 1, 2, and 3 in the figure. This lets our training code obtain architecturally valid references to them in the linked list nodes. For this subsection, we assume that adjacent to these strings

is another JavaScript string that the attacker’s webpage does not own. As such, JavaScript from the attacker’s webpage cannot reference this string, making it architecturally out-of-bounds.

After making the last node’s data variable reference Str 3, we traverse the linked list with the JavaScript code shown in Listing 4.11. While similar to the native linked list traversal, Line 3 is the key difference. As we cannot dereference pointers in JavaScript, we retrieve `n.data` instead. By doing so, under the hood, Safari’s JavaScript engine dereferences the addresses of the string objects which we have allocated in strides, thereby training the LAP. Then, to access the contents of the out-of-bounds string, we call the `charCodeAt` method, which returns the ASCII value of the character at the given index. Finally, for the `transmit` function in Line 4, we encode the bits of the ASCII value into the cache state, such that we can recover them subsequently using the amplifier from subsection 4.6.1.

Listing 4.11: Our browser-based gadget to mistrain the LAP using the data structures shown in Figure 4.14.

```
1 let n = first_node;
2 while (n) {
3     const secret = n.data.charCodeAt(index);
4     transmit(secret);
5     n = n.next;
6 }
```

4.6.3 Exploiting Safari’s Memory Model

Thus far, we use the linked list training method on contiguously allocated JavaScript strings for an out-of-bounds memory access, which will proceed through speculation as long as the memory layout is also that of a string. We also amplify cache hits and misses to a point where they can be distinguished with coarse timers. Now, we recall from subsection 4.4.5 that the LAP’s maximum stride is 255 bytes. We indicate this as the red zone beyond our training strings in Figure 4.15: in contrast, data placed further than that in the black zone

are not exploitable. Hence, our goal is to place sensitive data from a target website into this span.

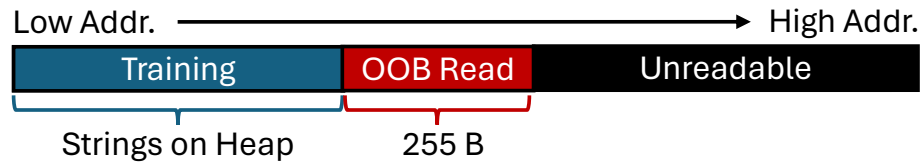


Figure 4.15: Diagram of the regions of WebKit heap memory that are and are not reachable with our attack primitive.

Consolidation and Object Heaps. WebKit uses its own memory allocator, `libpas`, separately from the operating system’s `malloc` API [114]. With `libpas`, WebKit partitions its virtual address space into several heaps, where each heap can be multiple disjoint areas of addresses but always contains JavaScript objects of the same type [115]. On the other hand, the iLeakage attack observed that WebKit does not isolate webpages into separate rendering processes when using the `window.open` API call [98, Section 5.1]. We confirm this observation. Moreover, we observe that JavaScript objects from different webpages but of the same type can share a heap (in addition to a process).

Inspecting WebKit’s Heap Allocations. Now, we inspect what data are allocated within the 255-byte reach of our training strings in the JavaScript string heap. We first spawn a new WebKit rendering process to handle the attacker webpage. Then, we use the `window.open` call to render the target webpage in the same address space. Lastly, we allocate our training strings and inspect the address space using a debugger. However, we observe that strings from the target webpage’s DOM are not within the reach. While we do find reachable memory where WebKit has allocated a JavaScript string, the strings do not contain any user data. Instead, they are internal to WebKit’s built-in JavaScript APIs, such as error messages. We depict this case in the memory diagram of Figure 4.16 (Top).

Massaging with Memory Pressure. Next, we observe that memory pressure causes `libpas` to behave differently, allocating more virtual address ranges for the string heap. Hence, we allocate ‘filler’ JavaScript strings in the attacker webpage first to exert memory

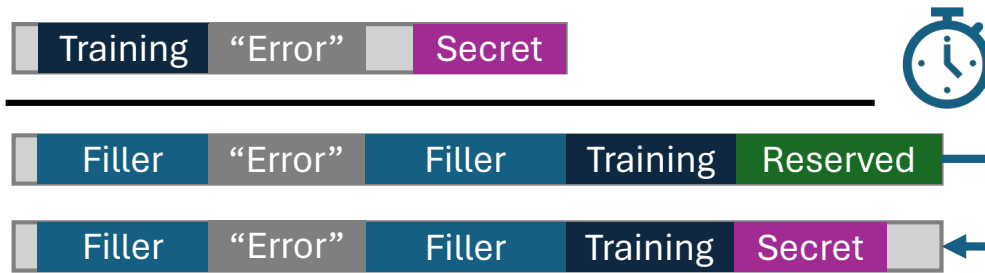


Figure 4.16: Simplified memory layouts of WebKit’s address space. (Top) No memory pressure is applied. (Middle) Memory pressure is applied with the filler strings, and the target page has not loaded yet. (Bottom) Memory pressure is applied, and the target page has completely loaded.

pressure. Then, we allocate our training strings. At this point, we observe that the memory contents of the 255 bytes adjacent to the last training string are vastly different. Instead of containing internal WebKit strings, the region becomes reserved for future string allocations. See Figure 4.16 (Middle). Next, we load the target page using `window.open`. As the page renders and executes JavaScript, the reserved region becomes populated with strings as the target page’s scripts interact with its DOM. Accordingly, we can probabilistically induce the allocation of strings holding sensitive information within the 255-byte reach of the LAP, as shown in Figure 4.16 (Bottom). Finally, we note that memory pressure can be applied discreetly and without attracting the user’s attention, as the attacker’s website only needs to allocate a few megabytes of filler strings to trigger `libpas` into creating these reserved areas.

4.6.4 Reading Data Across Websites

With the setup for SLAP now being complete, we now demonstrate its application to real-world targets. We first set up a proof-of-concept target page that continuously queries a server for the current time and displays it to the user when the response arrives. In order to display the updated time, our website must modify the DOM. This causes WebKit to scan all DOM nodes and store the resulting HyperText Markup Language (HTML) as a string, reliably putting it into the LAP’s reach when the attacker webpage `window.open`s our

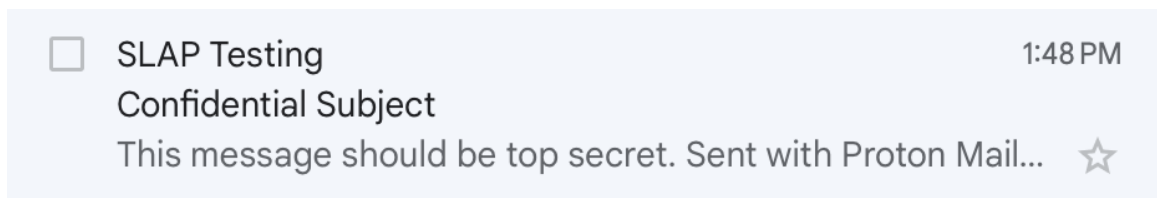
target page.

Initial Benchmarks. Using the M2 CPU, we run SLAP end to end on macOS and Safari. Firstly, our attacker page allocates 100,000 filler strings. Each string’s inline size on the heap is 16 bytes, resulting in 1.6 MB of pressure on `libpas`’s heap for JavaScript strings. Secondly, we allocate the training strings for the SLAP gadget, observing that about 500 of them can be contiguously allocated. Thirdly, we open the target page described above and observe that the DOM is usually 64 bytes away from the last training string. Hence, to train the LAP on a 64-byte stride, we construct the linked list from every fourth string, resulting in 125 linked list nodes. Finally, we repeat reading the DOM of the target webpage 10 times, with a median bitwise accuracy of 87.9% and throughput of 0.384 bits per second. Here, we observe that most noise is from single-bit errors, and can be further filtered with repeated sampling.

Reading Inbox Data from Gmail. We now target real-world websites. JavaScript runs on virtually every website nowadays, interfacing with the `document` API to read and write parts of the DOM. As such, we now investigate which parts of the DOM of one of the largest email services can be allocated adjacently to our training strings.

We assume the target is authenticated to Gmail, and visits the attacker webpage. The attacker webpage allocates 1.7 MB of filler and training strings, and then calls `window.open` on Gmail’s inbox page when the mouse cursor is placed over itself. As Gmail loads, JavaScript in the page starts rendering the inbox, whose content is personalized to the target. Over repeated trials, we show that the subject line and the sender’s identity can land in the reachable out-of-bounds region of the LAP, allowing for recovery by the adversary in Figure 4.17.

Fingerprinting Amazon and Reddit Activity. The main pages of web services that recommend content to authenticated users act as fingerprints for their past activity. As such, we turn our attention to Amazon’s ‘Buy Again’ page and Reddit’s home page, as both are major e-commerce and forum platforms where users can express interest for certain product

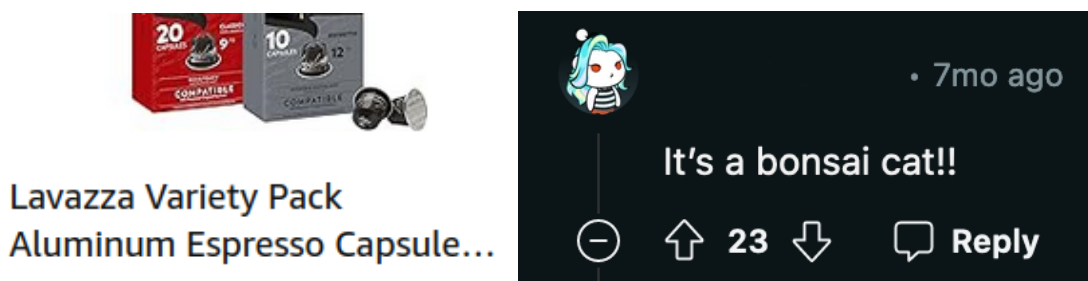


ZLAP Te{ting | Confiden~ivl SuBne#t

Figure 4.17: (Top) Email subject and sender name shown as part of Gmail’s DOM. (Bottom) Recovered strings from this page.

categories or discussion topics (‘subreddits’).

Similarly to the Gmail scenario, we assume the target is authenticated to Amazon and Reddit and visits our webpage. We repeat the attack procedure from before, but cause the `window.open` calls to open Amazon’s ‘Buy Again’ page and Reddit’s home page. Then, we observe what personalized data appears within reach of the LAP. On Amazon, we recover a product description of coffee pods, which the user has purchased previously, as we show in Figure 4.18 (Top Left, Bottom Left). Furthermore, on Reddit, we recover text from a comment on a post which belongs to a subreddit that the user subscribes to: see Figure 4.18 (Top Right, Bottom Right). Here, we note we can use Reddit’s search feature to recover the original post and subreddit.



Lava~ze VarigtyiPack Alwminum Es`resso!Capsuh! | Itâs a bonsai\$cat%!

Figure 4.18: (Top Left) A listing for coffee pods from Amazon’s ‘Buy Again’ page. (Bottom Left) Recovered item name from Amazon. (Top Right) A comment on a post on Reddit, and (Bottom Right) the recovered text.

4.7 Mitigations

Having demonstrated new attack surfaces opened by the LAP, we now reason about strategies to mitigate them. As a first step, in subsection 4.4.1, we observed that E-cores on the M2 and M3 CPUs do not have the LAP. On MacOS and iOS, this entails a simple user code change in the form of adding a call to the `pthread_set_qos_class_self_np` API with the `QOS_CLASS_BACKGROUND` argument [116]. However, this approach is not universal: it may be undesirable to computation-intensive applications, and E-cores in future Apple CPUs may or may not have the LAP.

Attempting to Disable the LAP. Next, we seek to disable the LAP altogether by writing to several candidate system registers. On the M2 and M3 CPUs, our first candidate is the Data Independent Timing (DIT) bit of the Armv8.4-A ISA [117], which specifies that certain instructions should have a latency independent of the operands when this bit is set. Furthermore, the DIT bit can be set from userspace (EL0) and per-process on macOS, and has been shown to disable the Data Memory-dependent Prefetcher (DMP) on the M3 CPU [22]. Hence, we hypothesize that it may also disable other unconventional performance optimizations such as the LAP. For the same reason, our second candidate is bit 30 of the `HID11_EL1` system register, which has been shown to disable the DMP on the M1 and M2 CPUs [22, 118]. However, this bit can only be set on Linux, which has no M3 support at the time of writing.

Testing More System Registers. On the M2 CPU and Linux, we reference the Asahi project’s documentation on Apple Silicon’s system registers [119] for bit descriptions pertaining to prefetching. From this, we test the following five bits: Bit 13 of `HID2_EL1` “Disable MMU MTLB Prefetch”, Bits 44-45 of `HID5_EL1` “Disable HWP Load/Store”, Bit 0 of `HID10_EL1` “Disable HWP Gups”, and Bit 3 of `ACTLR_EL1` “Disable HWP”. Here, we hypothesize that ‘HWP’ is an acronym for hardware prefetcher.

Results. We repeat the experiments with a loop of RAW-dependent loads from subsec-

tion 4.4.1 with one candidate bit set at a time, with all others at their default values. The speedup on the M2 and M3 CPU’s P-cores is still present with the DIT bit set, indicating it does not disable the LAP. For the system registers, the `HID5_EL1` bits cause Linux to crash immediately. All other bits do not affect operation, but fail to eliminate the speedup likewise.

The ‘Kill Switch’ Bit. Another system register bit of interest on Apple CPUs is bit 4 of `HID4_EL1`. However, its description is “Force CPU Oldest In Order” [118], resembling a kill switch to disable all out-of-order execution. Indeed, we do not observe a speedup from the experiments in subsection 4.4.1 and subsection 4.4.2 after setting this bit. Repeating the misprediction experiment from subsection 4.4.3, we observe only dummy values over the covert channel and no secret values, confirming again that this bit does disable the LAP.

However, we cannot recommend this as a mitigation due to macOS currently lacking support for setting this bit even for privileged users, necessitating Apple-released software updates. Moreover, the performance penalty is dire: we measure this with PassMark’s PerformanceTest version 11.0.1002. On default settings, the M2 scores 8,098 points. However, with the ‘kill switch’ bit set, the score drops to 2,873, a slowdown of 2.81x. For comparison, this is just below the 2,929 points scored by an Intel Core i5-2500T, a consumer desktop CPU released in 2011 [120]. We leave the task of finding a method to disable the LAP at the CPU level without significant performance impact to future work.

Considerations for Safari. We emphasize the importance of site isolation [90], a mechanism preventing webpages of different domains from sharing rendering processes. Site isolation is already present in Chrome and Firefox [90, 91], preventing sensitive information from other webpages from being allocated in the attacker’s address space. While its implementation is an ongoing effort by Apple [96, 121], site isolation is not currently on production releases of Safari. On the contrary, we also reflect on `libpas`’s heap layout from subsection 4.6.3, allowing sites to not only share processes, but also heaps. Partitioning JavaScript heaps by at least several memory pages per-webpage would prevent

JavaScript strings from the target webpage from being allocated within the 255-byte reach of the LAP.

Future Mitigations. In the short term, a practical mitigation from Apple would be to disable the LAP when the DIT bit is set, following its precedent with the DMP. Doing so would allow developers to disable the LAP in sections of code handling secrets without affecting other programs (since its state is kept per-process), and subsequently re-enable it for non-sensitive code regions to leverage the LAP’s performance benefits on data dependencies.

For future microarchitectures, prior work in side-channel defense literature [122, 123, 124] proposes hardware support for marking memory regions or operands as holding secrets, and for speculation to abort if the CPU ever accesses marked data. We acknowledge this would make exploitation more challenging, since speculation would terminate before our gadgets can retrieve a marked secret and encode it over microarchitectural covert channels. However, this method requires vendors to release new hardware with extensive additions to circuitry, while requiring developers to update their software to explicitly mark sensitive data.

4.8 Conclusion

In this chapter, we discover a new mechanism for data speculation in the form of load address predictors in recent Apple CPUs. The LAP can issue loads to addresses that have never been accessed architecturally and transiently forward the values to younger instructions in an unprecedentedly large window. We demonstrate that, despite their benefits to performance, LAPs open new attack surfaces that are exploitable in the real world by an adversary. That is, they allow broad out-of-bounds reads, disrupt control flow under speculation, disclose the ASLR slide, and even compromise the security of Safari. In the landscape of the decline of Moore’s Law birthing more exotic microarchitectural optimizations, we believe that LAPs may not be an Apple exclusive, either now or soon. As such,

we emphasize the need for novel hardware and software countermeasures against LAPs in future work.

CHAPTER 5

FLOP: BREAKING THE APPLE M3 CPU VIA FALSE LOAD OUTPUT PREDICTIONS

To bridge the ever-increasing gap between the fast execution speed of modern processors and the long latency of memory accesses, CPU vendors continue to introduce newer and more advanced optimizations. While these optimizations improve performance, research has repeatedly demonstrated that they may also have an adverse impact on security.

In this work, we identify that recent Apple M- and A-series processors implement a LVP, an optimization that predicts the contents of memory that the processor loads before the contents are actually available. This allows processors to alleviate slowdowns from Read-After-Write dependencies, as instructions can now be executed in parallel rather than sequentially.

To evaluate the security impact of Apple’s LVP implementation, we first investigate the implementation, identifying the conditions for prediction. We then show that although the LVP cannot directly predict 64-bit values (e.g., pointers), prediction of smaller-size values can be leveraged to achieve arbitrary memory access. Finally, we demonstrate end-to-end attack exploit chains that build on the LVP to obtain a 64-bit read primitive within the Safari and Chrome browsers.

5.1 Introduction

The computer industry is witnessing an ongoing paradigm shift in the desktop and server markets, where x86-based Intel and AMD CPUs are now competing with a growing portfolio of Arm-based CPUs from Apple, Qualcomm, and Amazon. As new contenders in the market, many of these heavyweight Arm CPUs are notable for being clean-sheet designs, differing significantly not only in instruction set but also in microarchitecture from x86

CPUs.

However, clean-sheet designs are not free of clean-sheet problems. For instance, Apple’s M-series CPUs have been reported to exhibit security issues pertaining to novel CPU features such as data-dependent prefetching [21, 22], pointer authentication [52], and instructions that allow for timerless cache attacks [125], in addition to more traditional issues that have also plagued x86-based CPUs such as speculative execution [78, 98] and data-dependent throttling [33].

In a never-ending quest to improve performance, architectures also have attempted to streamline the execution of dependent instructions. One such proposed optimization aims to alleviate slowdowns from Read-After-Write (RAW) dependencies. These occur when younger instructions read from the same location that an older instruction writes to, forcing the CPU to serialize these instructions. Accordingly, several works propose mechanisms that de-serialize RAW dependencies through prediction [99, 100, 101, 102, 103, 104, 108, 126, 127, 128], resulting in performance speedups.

With Spectre [1] and followups [5, 6, 11, 14, 15, 17, 31, 52, 98] demonstrating the security hazards of predictions, in this chapter we ask the following question:

How are RAW dependencies handled on emerging CPU designs? What optimizations do they entail and what are their security implications?

5.1.1 Our Contributions

In this chapter, we discover a prediction mechanism for RAW dependencies that, to the best of our knowledge, was previously unseen in the wild. We show that Apple’s M3, M4, and A17 Pro CPUs all optimize RAW dependencies via a load value predictor (LVP), which observes data values returned from load operations. If the values are constant, these CPUs can open a speculation window the next time this load executes, rather than waiting for the result to become available after a RAW dependency resolves. Within the speculation window, the predicted load values can be forwarded to arbitrary younger instructions that

depend on them, thus causing the CPU to compute on the predicted value under speculation.

After characterizing Apple’s LVP implementation, we proceed to demonstrate its security implications especially when the mispredictions cause transient computation using stale load values. Here, we show how the LVP causes type confusion attacks in Safari and hijacks control flow in Chrome. In both cases, we package our primitives into end-to-end attacks that read arbitrary 64-bit addresses, allowing us to recover sensitive data across webpage origins.

Discovery and Characterization of the Apple LVP. We begin with an experiment that serially reads from a randomly shuffled collection of memory addresses and measures the running time. Remarkably, despite serializing the instruction via a RAW dependency, the M3 CPU runs drastically faster when the data stored in our memory addresses is a constant, compared to when it is randomly generated. Next, we proceed to reverse engineer the LVP’s activation criteria, finding that the Apple LVP activates only for constant load values as opposed to striding ones, and keeps training state per instruction address for up to 72 different addresses. Moreover, it predicts arbitrary values only for 4-byte-wide loads and smaller, and not for 8-byte loads (which could be pointer values).

Subsequently, we demonstrate a primitive that causes the LVP to mispredict and speculatively compute on a stale value. Here, 250 training loads suffice to train the LVP with enough confidence for reliable mispredictions, with the resulting speculation window lasting up to 330 cycles. Next, we add a layer of indirection, causing the stale value to select and dereference an incorrect pointer, or even perform incorrect function calls. Thus, we demonstrate that LVP-induced speculation violates memory safety not only with 64-bit out-of-bounds reads, but also by diverting execution to rogue functions that are never architecturally invoked. Finally, we also show that the LVP can be mistrained in kernel-space assuming the existence of suitable gadgets, exacerbating the security risks.

Exploiting LVP Mispredictions in Safari. Going beyond LVP activations in the OS kernel, we show the practical security implications of the LVP by demonstrating 64-bit

out-of-bounds reads in Safari, despite Apple’s recent hardening attempts. To that aim, we use a gadget that accepts an object, which has a string member. The gadget dereferences the 64-bit pointer to the backing store of the string, and transmits the value it reads through a cache covert channel. In execution, we pass an array that contains raw binary data instead of the expected object type, but we exploit the LVP to confuse the processor about the input type.

Consequently, the CPU transiently executes the gadget on the wrong object type. Thus, instead of dereferencing a pointer to a string, the processor uses the attacker-controlled data in the array, confusing the CPU to read from an address of the attacker’s choosing and transmitting the read value through a covert channel. With this speculative type confusion primitive, we run the FLOP-Data attack end-to-end, recovering the target’s location history from Google Maps, inbox content from Proton Mail, and events stored in iCloud Calendar.

Exploiting LVP Mispredictions in Google Chrome. Going beyond disrupting data flows, we present the FLOP-Control attack, which causes the CPU to execute the wrong WebAssembly function under speculation in Google Chrome. In this attack, we call functions indirectly from a dynamic function dispatch table. Firstly, we mistrain the LVP on the table index such that the CPU retrieves the wrong dispatch object. Then, we mistrain the LVP again in a nested manner, such that the CPU will mistakenly validate the function arguments against the dispatch object when they are in fact invalid. This results in the CPU branching to a rogue function, which then computes on unchecked arguments.

When this nested LVP misprediction happens, architecturally, we provide a 64-bit integer argument to a function which takes a 64-bit integer. However, the LVP causes control flow under speculation to be misdirected to a function taking a `struct` reference, which is implemented as a 64-bit pointer in Chrome. Therefore, we cause the CPU to confuse data with an address, leaking the target’s credit card information and billing address on Square storefronts.

LVP and DIT Interaction. Finally, we discover that the Arm ISA’s Data Independent

Timing (DIT) bit disables the LVP on the M3 CPU on a per-process basis, with no privileges needed to set the bit. See section 5.7 for details.

Summary of Contributions. We contribute the following:

- We identify that an LVP exists on recent Apple CPUs, reverse engineering the pre-conditions for activation. We demonstrate a gadget to cause load value mispredictions, showing how the LVP can cause loss of control flow and data integrity under speculation (section 5.4).
- We weaponize the LVP to perform a speculative type confusion attack on Safari, escaping its sandbox and recovering sensitive data from popular web services (section 5.5).
- We weaponize the LVP again to transiently execute the wrong function in Chrome, demonstrating another sandbox escape and recovery of secrets (section 5.6).

Responsible Disclosure. We disclosed our results to Apple’s Product Security Team on September 3, 2024. Apple has acknowledged our disclosure and is continuing to investigate our report. For more details, see section 5.9.

5.2 Background

Cache Side-channel Attacks. Virtually all modern CPUs use caches, which are small data buffers on or close to the CPU cores. For data that is used recently or frequently by the cores, the cache reduces memory access latency. However, this also means an adversary on the same system or core can time accesses to data and gain information about a target program’s memory access patterns. Previous works demonstrate several techniques to do so, with some examples being Flush+Reload [54, 55] and Prime+Probe [24, 25, 56, 57, 59, 60, 62].

Out-of-Order and Speculative Execution. Another ubiquitous performance optimization beyond caches is speculative and out-of-order execution, which allows processors to

deviate from program-induced instruction order, particularly when arguments are not readily available. When an execution reaches a branch whose outcome cannot be immediately resolved, the CPU attempts to generate predictions based on prior behavior, and continue executing instructions down that path until the correct control flow can be computed. However, this implies that CPUs can transiently execute the wrong instructions or operands. The Meltdown [2] and Spectre [1] attacks pioneer the insight that such transient execution carries grave security consequences, due to microarchitectural state not being completely reverted in case of mispredictions. This has since resulted in numerous followup, breaking nearly all hardware-backed security domains [3, 5, 6, 11, 14, 15, 16, 17, 31, 52, 63, 65, 66, 67, 68, 70, 72, 73, 98, 129].

Read-After-Write Dependencies. In a never-ending quest to improve performance, computer architectures also have attempted to streamline the execution of dependent instructions. While most dependencies can be solved via on-the-fly register renaming, Read-After-Write (RAW) is a fundamental data dependency for all pipelined CPUs when an older instruction writes to the same location that is read from by a younger instruction. Typically, the CPU must run the older instruction to completion before the younger one can execute, because its operand cannot be computed before that point. Accordingly, RAW dependencies result in a slowdown of the pipeline, prompting computer architects to propose value prediction [99, 100, 128, 130, 131]. After observing values from instructions repeatedly, a CPU with value prediction can speculate past RAW dependencies by executing younger instructions with the predicted value.

Load Value Prediction. Within value prediction, Load Value Predictors (LVP) are among the most proposed [101, 102, 103, 104, 108, 132, 133], because not only do loads (and stores) occur frequently in program code, but they vary in latency greatly from <10 cycles for L1 cache hits to hundreds of cycles from main memory. Mitigating this, Figure 5.1 presents an outline of a typical LVP mechanism.

We begin at ① in the figure, depicting a loop with a RAW dependency. That is, the next

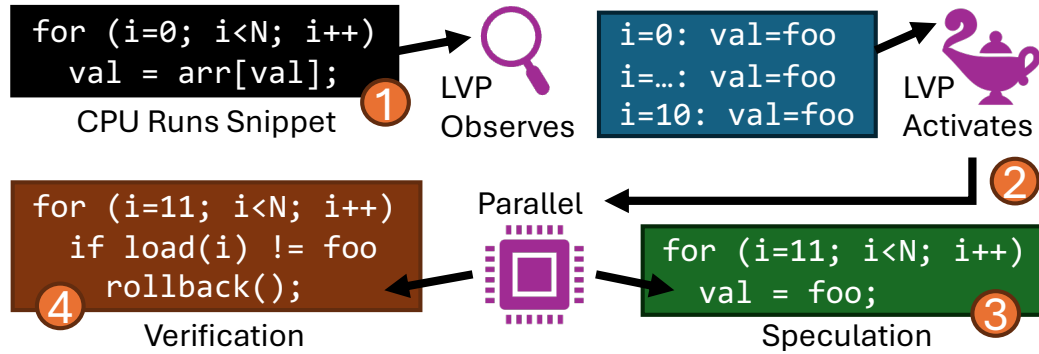


Figure 5.1: Overview of load value prediction.

index into the array to load from cannot be determined until the previous load completes and its value is saved into the `val` variable¹. The LVP observes values being returned from the loads, oftentimes from the same instruction address as is the case for loops. If the loads predictably return the same value (such as `foo`) more than a set number of times as indicated in ②, the LVP activates.

Once activated, the LVP causes the CPU to perform two tasks, ③ and ④, in parallel. The first task ③ is to speculatively run the rest of the code using the predicted load value `foo` for the `val` variable, bypassing the loads caused by indexing into the array. The second task ④ is to execute the loads one by one, where we denote `load(i)` as the value returned from the *i*-th load in the loop. If all of them equal `foo`, the results of the first task are committed to the CPU’s architectural state. Conversely, if there is a mismatch (e.g., from another core modifying the array), the speculative results are reverted and execution is replayed at `i=11`. Therefore, ③ can proceed until ④ either completes or rolls back.

5.3 Threat Model and Setup

We assume a typical browser-based threat model for our attack in section 5.5 and section 5.6, where the target user runs a web browser and visits an attacker-controlled web-page. Furthermore, we focus on recently released Apple CPUs. Here, we assume the target

¹We note that this pointer-chasing scheme is in fact a RAW dependency in the CPU’s register file, even though main memory is not written to.

system is a Mac with an Apple silicon CPU running macOS 14.5, Safari 17.5, and Chrome 128.0 with out-of-the-box settings. All versions are the most recent at the time of writing. Finally, we do not modify settings of side-channel countermeasures, leaving them in their default state.

5.4 Analysis of the Apple M3 LVP

5.4.1 Ascertaining the LVP's Presence

We now describe our procedure to test for load value prediction. Initially, we allocate a large buffer called `mem` that spans multiple cache lines. For our experiment, we fill this buffer with a constant value such that the value of a load from anywhere in the buffer will be predictable. The pseudocode in Listing 5.12 operates on this `mem` buffer.

Listing 5.12: Our routine to measure the memory access latency on randomly shuffled addresses, where the load values may be identical or random.

```
1  memset(mem, CONST, MEM_SIZE);
2  int offsets[ITERS] = getOffsets(mem, ITERS);
3  shuffle(offsets);
4  flushBuffer(mem);
5  uint64_t start = getCycles(), junk = 0;
6  for (int i = 0; i < ITERS; ++i)
7      junk = junk + *(mem + offsets[i]);
8  uint64_t end = getCycles();
9  return end - start;
```

We measure the number of CPU cycles to access several random cache line-aligned offsets (multiples of 128 bytes) comprising the `mem` buffer, where `ITERS` is the number of offsets accessed. To this aim, Line 1 `memset`s the entire buffer such that any byte-wide load from `mem` will return `CONST`. Line 2 computes the aligned offsets, storing them in the `offsets` array. Then, Line 3 randomly shuffles the offsets, making the access pattern unpredictable to avoid activating hardware prefetchers. Line 4 flushes all of `mem` from the cache. By doing so, we ensure each access is a cache miss, creating favorable conditions for an LVP to activate (if one exists) to alleviate the slowdown in performance. After Line

5 obtains a timestamp, Lines 6-7 perform byte-wide loads to the offsets into `mem` in a loop. Here, to cause the loads to run serially, we insert a RAW dependency as the `junk` variable. Finally, Line 8 takes the second timestamp, and Line 9 returns the elapsed cycles.

We compare this experiment against our control, where we replace the `memset` in Line 1 with filling randomly generated values into the `mem` buffer such that the load values from `mem` will be unpredictable instead of constant. Figure 5.2 shows a graphical representation of the load addresses and values handled by Lines 6-7 in our experiment and control.

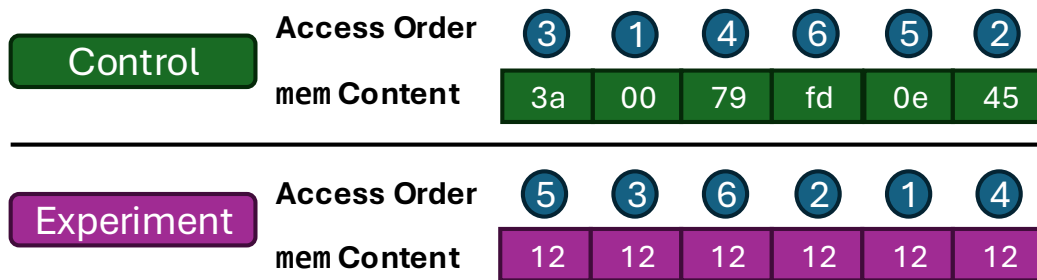


Figure 5.2: Comparison of our experiment against its control. While the access order for load addresses is always random, we vary the content of the `mem` buffer to be constant in our experiment and random in our control.

Experimental Setup. For precise measurements, we need kernel-level support for counting CPU cycles, pinning programs to CPU cores, and flushing cache lines. As these features are not available on macOS by default, we use Apple’s Kernel Debug Kit (KDK) for macOS 14.5 build 23F79 to run macOS with the KDK’s development kernel, and specify the details in Appendix D. With this setup, we run the control and experiment on the Apple M2 and M3 CPUs. All modern Apple CPUs have heterogeneous core designs, packaging a combination of high-performance P-cores and energy-efficient E-cores. Therefore, we consider each core type separately since they vary significantly in microarchitecture. We vary the `ITERS` parameter in increments of 10 from 10 to 500, and we use the median of 100 samples for each data point.

Results. Figure 5.3 shows the resulting plots. On the M2’s P-cores (top left), our experiment does not result in particularly faster runtimes compared to our control. That

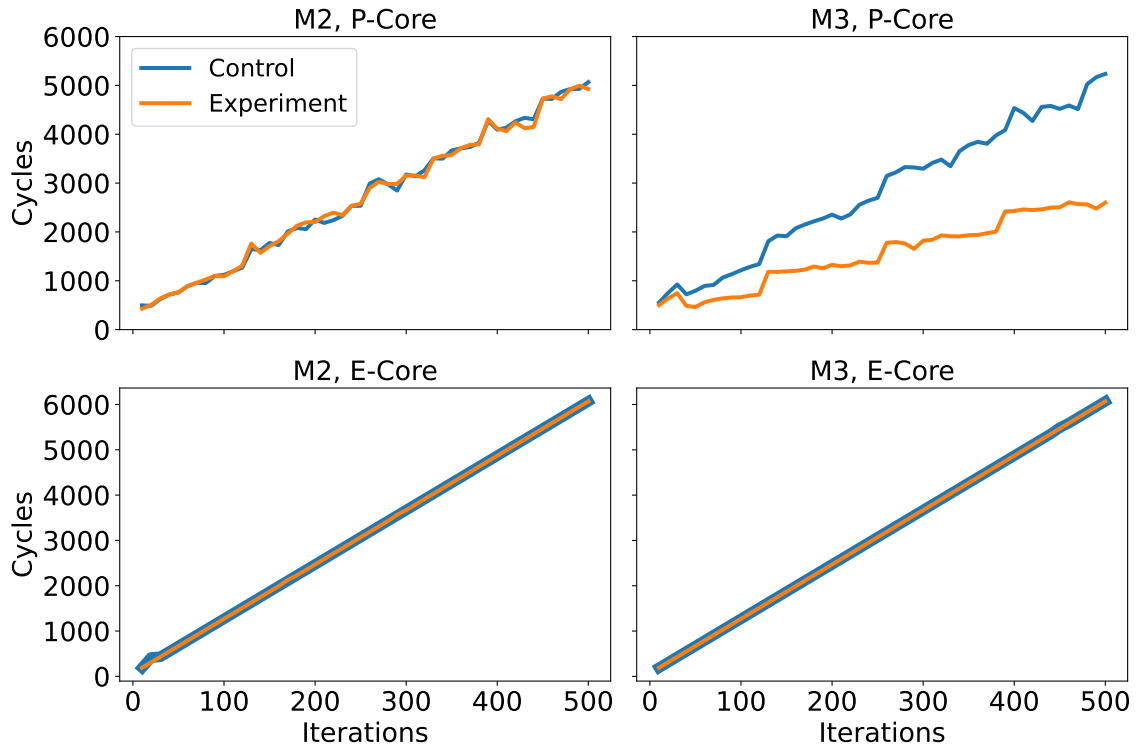


Figure 5.3: Runtimes of the gadget in Listing 5.12 on each CPU and core type. The control and experiment plots heavily overlap for the E-cores.

is, the runtimes increase linearly with the number of loads performed, due to the RAW dependency between instructions, and does not depend on the values loaded from memory being randomized or constant. Thus, we conclude that an LVP is not present. In contrast, the runtimes from the M3’s P-cores (top right) show a significant speedup from 40 iterations and above on our experiment when the load values are constant. At the maximum of 500 iterations, our workload only takes about half the cycles in that case compared to when the load values vary randomly. As such, we determine the P-cores are indeed equipped with an LVP, alleviating the performance penalty from RAW dependencies. Moving on to the E-cores of both CPUs (Figure 5.3 bottom), these generate plots that do not differ significantly by load value, indicating the lack of LVPs.

LVP Presence on Other Apple CPUs. After determining that the LVP is present on the M3 and not the M2, we seek to extend our experiments to more Apple CPUs. To that aim, we create a portable version of Listing 5.12 by compiling it to WebAssembly. Next, we fix

the `ITERS` variable to 10 million to make the runtime difference measurable using Safari’s coarse timer (1 ms). From this, we indicate whether each CPU has an LVP on Table 5.1. That is, the LVP is present on the M3, M4, and A17 and absent on the M2, A15, and A16.

Table 5.1: Survey of LVP presence on recent Apple desktop and mobile CPUs.

Apple CPU	Tested Device	Has LVP?
M2	MacBook Air (A2681)	No
M3	MacBook Pro (A2918)	Yes
M4	iPad Pro 7th Gen. (A2926)	Yes
A15 Bionic	iPhone 13 Mini (A2481)	No
A16 Bionic	iPhone 14 Pro Max (A2651)	No
A17 Pro	iPhone 15 Pro (A2848)	Yes

5.4.2 Identifying LVP Activation Criteria

Having established the LVP’s existence, we now investigate its behavior for load instructions of different width, whether it detects striding load values (as opposed to constants) and its response to loop unrolling. We compare the outcomes of these changes against our initial M3 P-cores observations (top right of Figure 5.3), which is copied over and shaded in gray. Finally, we also identify how many predictions the LVP mechanism can manage simultaneously.

Width of Loads. We repeat the measurements from subsection 5.4.1, but change the pointer dereference in Line 6 of Listing 5.12 to have a load instruction of varying widths. We test for a speedup compared to our control when using 1, 2, 4, and 8-byte loads, and for each of the 100 samples we vary the constant value that every load returns. See Figure 5.4.

Remarkably, while all other load widths activate the LVP on any constant value fitting that width, we observe that activation on 8-byte wide loads occurs only when the load value is zero. We conjecture that this may be a countermeasure for memory safety such that the LVP will not learn values of pointers. That is, with the M3 being a 64-bit CPU, pointers are 8 bytes wide. Furthermore, on 64-bit macOS executables, any virtual address below

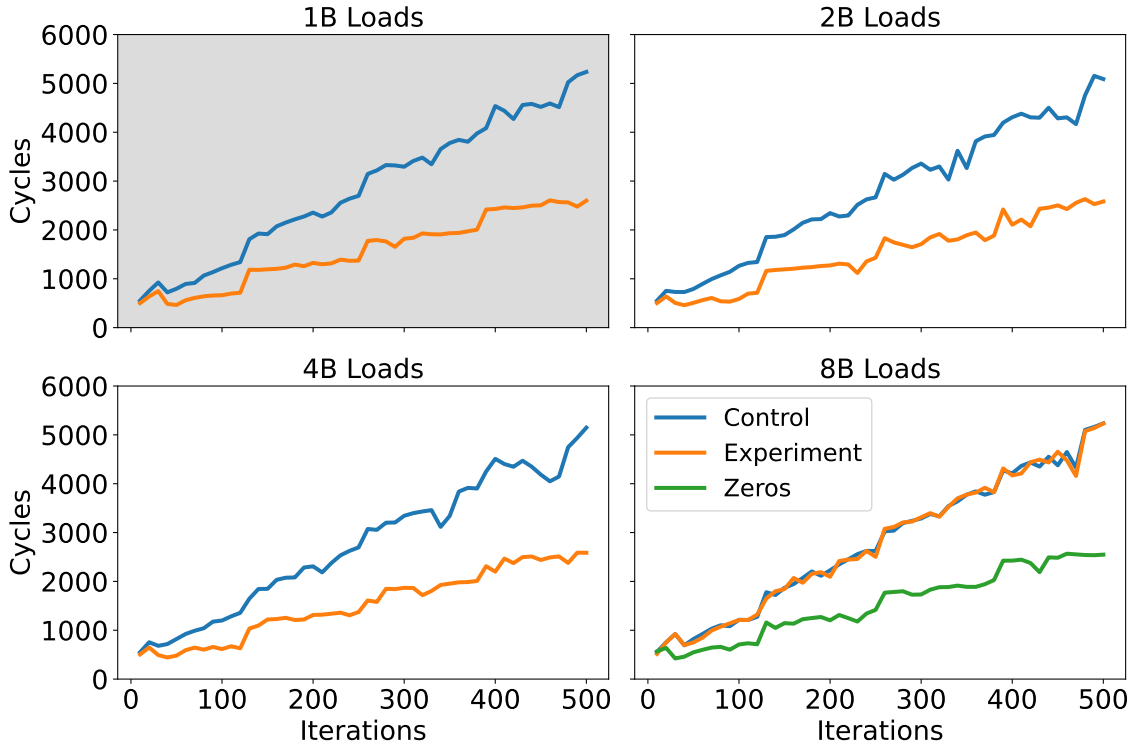


Figure 5.4: The effect of load width on LVP activation for the M3 CPU.

$0 \times 100,000,000$ is invalid [134].²

Striding Load Values. To test if the LVP learns load values that change with a fixed stride, we insert a subroutine in Listing 5.12. Between Line 3 where we randomize the access order and Line 4’s flushing of all cache lines in the `mem` buffer, we write values to `mem` incrementing from 0 in the order obtained from Line 3. As before with load widths, we repeat measuring but with striding load values on the right of Figure 5.5, and compare it with constant load values on the left.

Here, we observe that the runtime as the number of loads increases resembles the trend when the load values are random. As such, we conclude that Apple’s implementation of the LVP trains only on constants.

Loop Unrolling. Without modifying the source code from Listing 5.12, we direct the compiler to fully unroll Lines 6-7 to exactly `ITERS` different load instructions for each

²We note however that this strategy is not entirely safe, as LVI-NULL [73] demonstrates transient execution attacks using address zero.

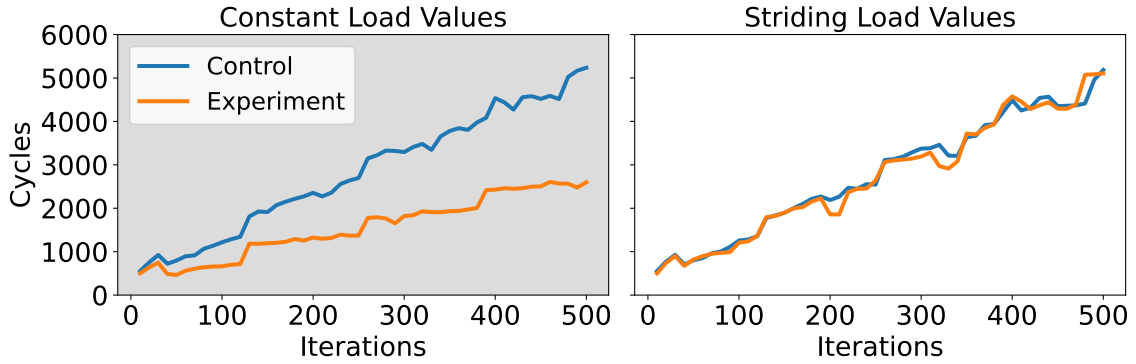


Figure 5.5: Runtime when the M3 CPU loads values that stride, instead of the same value on each load. Our control continues to load random values.

value of `ITERS` (10 to 500, in increments of 10). This causes a load placed at a particular instruction address to be executed only once per trial, as opposed to tens or hundreds of times. We plot the runtimes of our control and experiment on the unrolled binary to the right of Figure 5.6, and contrast them with the original loop on the left.

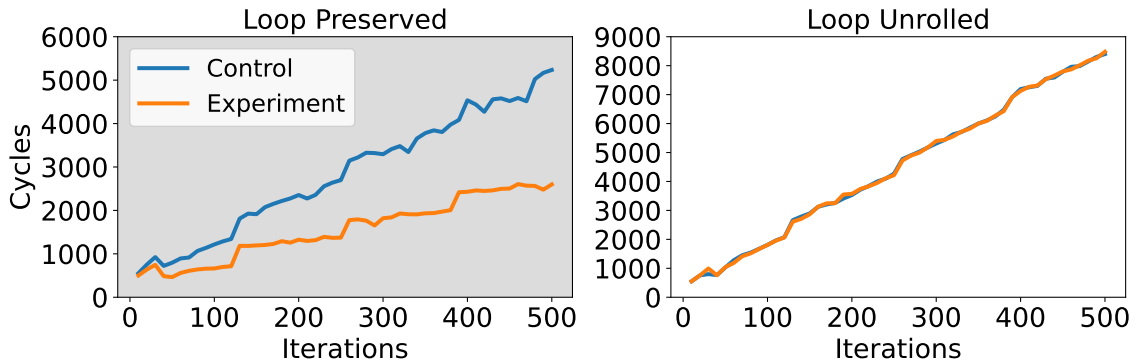


Figure 5.6: Effect of loop unrolling on LVP activation. We attribute the longer latency on the unrolled plot to the much larger binary size, as more instructions must be fetched by the CPU.

Notably, unrolling the loop causes the runtime on constant load values to scale nearly identically to our control with random load values, indicating that the LVP does not activate. Hence, we conclude that LVP trains with local scope (presumably with entries tagged with the instruction address) instead of training on a global window of recent load values.

Simultaneous Prediction Capacity. Lastly, after noticing the LVP’s local scope, we seek to determine how many distinct load instructions the LVP can track and activate on. To

that aim, we create several copies of the `mem` buffer from Listing 5.12, shuffling the access order individually. For the traversing loop in Lines 6-7, we copy the load instruction in Line 7 such that each line traverses a distinct clone of `mem`, and we also insert n dummy instructions between these loads to test if the LVP’s capacity is dependent on the instruction address bits (akin to set-associative caches). Then, we divide the timestamp in Line 8 by the number of load instructions to record the average traversal time. For each value of n , we increase the number of copies that simultaneously train the LVP until we observe the experiment’s average traversal time spike towards that of the control (indicating that the LVP failed to predict at least one copy). We plot the results in Figure 5.7.

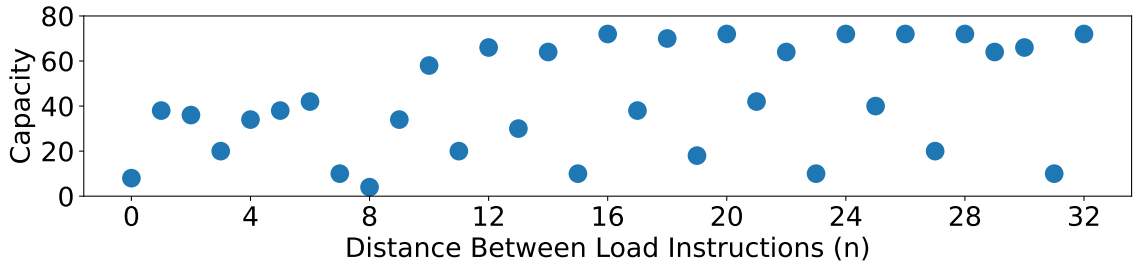


Figure 5.7: Maximum number of distinct load instructions the LVP can support before a timing spike is observed, depending on the instruction distance between them. In the 64-bit Arm ISA, every instruction is 4B wide.

Interestingly, we observe that the maximum number of copies the LVP can accommodate is 72, but this also depends on how far the load instructions are spaced apart. Rerunning the experiment with ASLR disabled, we observe the same result. Thus, we conjecture the LVP’s internal state cache may be 4-way set-associative (as the LVP could always predict at least four addresses regardless of the value of n) and uses a hash function on the page offset bits to determine the set.³

³Our attacks use the same instruction address to train and exploit the LVP. We leave the reverse engineering of this hash function to future work.

5.4.3 Measuring Mispredicted Load Values

Moving away from LVP-induced timing differences, we now investigate if the LVP uses its prediction to compute speculatively on arbitrary downstream instructions. Listing 5.13 outlines our experiment for measuring LVP speculation.

Listing 5.13: Code snippet for measuring mispredictions. We train the LVP on the load value `foo`, but then change the architectural value to `bar`. We cause the LVP to still operate on `foo` (which is now stale).

```
1  uint8_t gadget(int offset) {
2      return *(mem + offset);
3  }
4  // LVP training on load value foo
5  int offsets[ITERS] = getOffsets(mem, ITERS);
6  shuffle(offsets);
7  uint8_t foo = 0xca, bar = 0xfe, val = 0;
8  memset(mem, foo, MEM_SIZE);
9  for (int i = 0; i < ITERS; ++i)
10     gadget(offsets[i]);
11 // Make LVP mispredict bar as foo
12 memset(mem, bar, MEM_SIZE);
13 flushBuffer(mem);
14 val = gadget(offsets[0]);
15 frTransmit(val);
16 return frRecv();
```

Gadget Overview. First, we focus on the `gadget` function in Lines 1-3, where we perform loads from the `mem` via a function that is never inlined. This ensures that the LVP-training load in Line 2 is always at the same instruction address, following our results from subsection 5.4.2 which suggest Program Counter (PC)-tagging. Here, the `mem` buffer is identical to our experiments in subsection 5.4.1.

Training the LVP. Subsequently, Lines 4-10 constitute the training routine. After we initialize and shuffle the order of accesses into `mem` in Lines 5 and 6, we declare values for the `foo`, `bar`, and `val` variables in Line 7. Then, we call the `memset` function on the buffer with `foo` in Line 8, such that all loads from `mem` will result in `foo` being returned. Indeed, this is how we train the LVP in Lines 9 and 10 by invoking the `gadget` to load `foo`

several times, where we control the number of training loads with the `ITERS` parameter.

Misprediction with Stale Load Values. Next, we induce a misprediction in Lines 11-16. We call `memset` again but with the value `bar` in Line 12, changing the architectural load value from anywhere in the `mem` buffer to `bar`. Line 13, as before, flushes the whole buffer to induce the LVP to use its prediction instead of waiting for the load to resolve from memory⁴. We then call the `gadget` just once in Line 14.

This time when executing `gadget`, the CPU cannot retrieve the data for the load quickly, since it misses the cache. As the LVP has observed `foo` being returned from the load instruction in Line 2 before, the LVP uses `foo` as its prediction. Thus, `gadget` returns `foo` to be stored in the `val` variable in Line 14, and then in Line 15 we use `Flush+Reload` to encode `val` into the cache state such that we can recover it later. Next, at some later point, the CPU realizes the correct load value in Line 2 is `bar`, when it arrives from main memory. Hence, it rolls back the incorrect execution, returning `bar` into `val` and then encoding `bar` into the cache. Finally, in Line 16, we recover the encoded values, where we observe both `foo` (the stale load value) in addition to `bar`. See Figure 5.8.

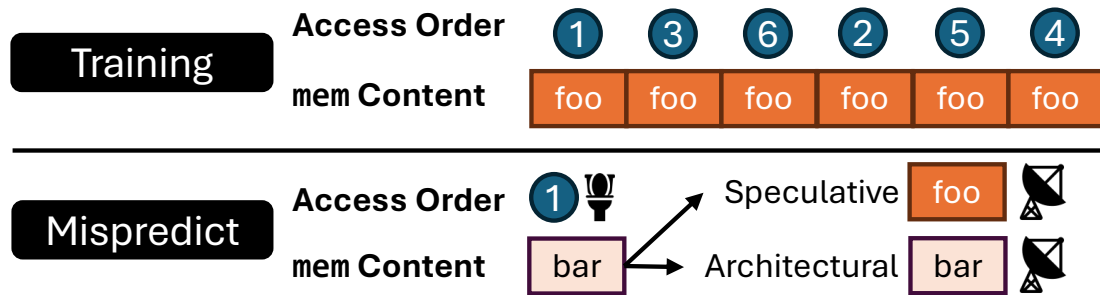


Figure 5.8: Graphical summary of the code in Listing 5.13. The LVP’s misspeculation and subsequent rollback causes `foo` to be transmitted, followed by `bar` (indicated by the satellite icons).

Measuring the Activation Threshold. With the above gadget, our initial goal is to observe mispredictions reliably. Thus, we seek to identify how many training loads are necessary on the M3’s P-cores. We vary the `ITERS` from 10 to 400 in increments of 2.

⁴To rule out the effects of predictive store-to-load forwarding, this flush operation is serializing, writing the stores from `memset` back to main memory.

For each value of `ITERS`, we run the code in Listing 5.13 1,000 times and plot the number of times we observe `foo` (the stale load value from the LVP) over the covert channel. Figure 5.9 plots the resulting misprediction counts.

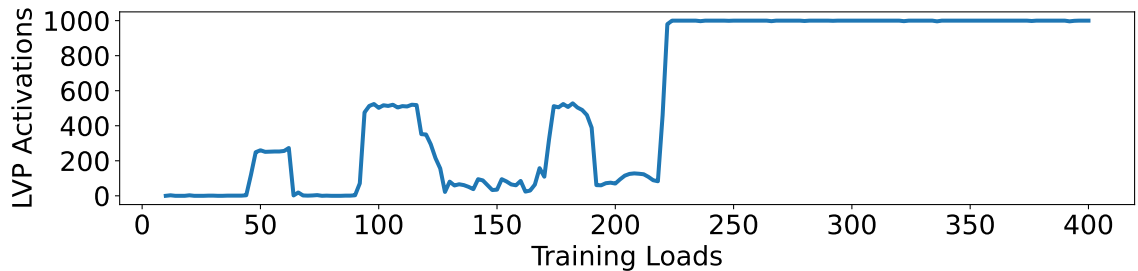


Figure 5.9: Effect of the number of training loads on the number of observed LVP activations with the stale load value (out of 1000).

We observe three spikes of activity before achieving reliable mispredictions. The first spike occurs around 60 loads, where the LVP activates about 25%. The second and third spikes occur around 120 and 180 loads respectively, with approximately 50% activation rate. Finally, past 240 loads, we observe activations with near-perfect reliability. Thus, in subsequent experiments, we train the LVP on 250 loads. Given that the previous spikes all occur around multiples of 60, we conjecture that the Apple LVP prefers training loads that are multiples of 60 until 240 loads, at which point it builds enough confidence to activate reliably regardless of training length.

Furthermore, above 240 loads, we observe that LVP activations are reliable even when thrashing the cache on all cores, leading us to conjecture that value-prediction state is stored in a dedicated microarchitectural buffer instead of cache lines.

Measuring State Persistence. Suppose we insert extra instructions between Lines 11 and 12 of Listing 5.13. Then, these instructions execute after LVP training completes, but before we run `gadget` once more to make the LVP mispredict. Hence, they allow us to test for conditions that cause the LVP’s internal state to either persist or attenuate. Thus, between Lines 11 and 12, we insert a busy-waiting loop of increasing duration to test for temporal conditions (such as a time-to-live field). In tandem, we test for the LVP’s

persistence with and without memory-intensive workloads, where for the latter we run stress-ng’s `vm` workload concurrently. We measure the number of mispredictions out of 100 trials in Figure 5.10.

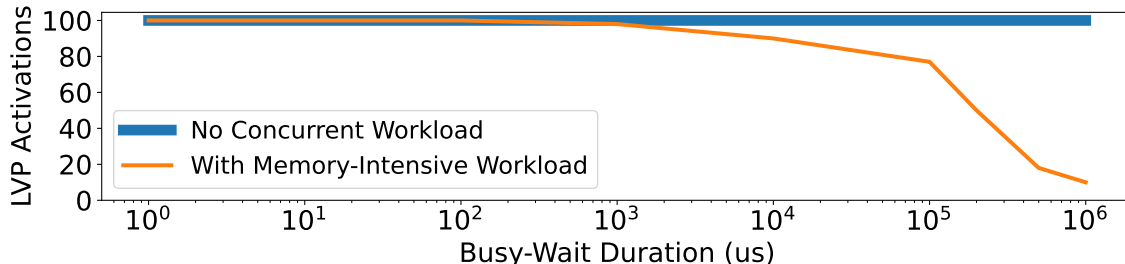


Figure 5.10: Number of observed LVP mispredictions when busy-waiting between training and misprediction, with and without a memory-intensive workload running on the same CPU core.

Observing Figure 5.10, we obtain reliable LVP activations regardless of the presence of stressors. However, the plots diverge at 10 ms, where the intensive load/store activity over time seemingly causes the activations to halve at 200 ms and decrease to one-tenth at 1 s. On the other hand, the LVP retains its internal state in the absence of stressors even after one-second busy waits (which, given the M3’s peak frequency of 4.05 GHz [135], is more than four billion cycles). From this, we hypothesize that the LVP’s state does not readily expire, but can be overwritten by memory activity on the CPU core over time. Furthermore, repeating this experiment with sending non-maskable interrupts instead of stress-ng, we identify that the LVP state is resilient to them after observing identical results to Figure 5.10 (Blue). On the other hand, if we replace the busy-wait with `sleep` to induce macOS to ‘park’ the CPU core in a low-power state, we observe that this resets the LVP state, resulting in no activations.

Measuring the Speculation Depth. Similarly to before, suppose we insert instructions between Lines 14 and 15 of Listing 5.13. During misprediction, the LVP transiently puts `foo` into the `val` variable in Line 14, and then transmits it over Flush+Reload in Line 15. Therefore, the inserted instructions become executed in the speculation window, allowing us to measure how long the LVP will compute on the predicted load value when the CPU’s

load target misses the cache. This time, we insert dummy `mul` instructions to keep multiplying the load value by 1 (in a data-dependent manner). We vary the number of `mul` instructions from 0 to 150 in increments of 5, and record the number of mispredictions out of 100 trials. In addition, we test for the LVP’s speculation depth when the CPU’s load target is cached by omitting Line 13 of Listing 5.13. We plot both speculation depths in Figure 5.11.

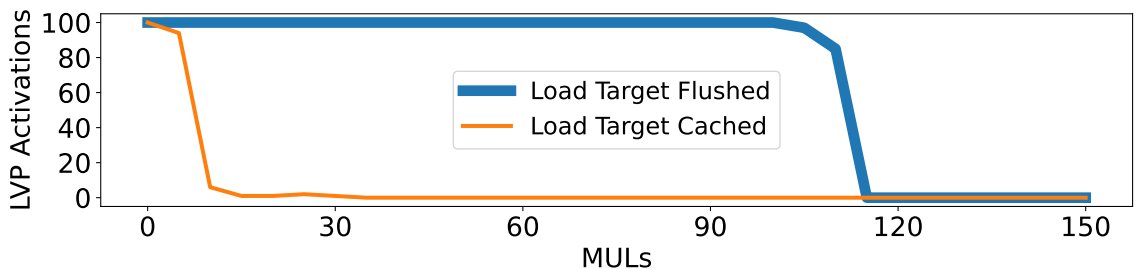


Figure 5.11: Number of observed LVP mispredictions when extra `mul` instructions are inserted between the speculative load and covert channel transmission. The speculation window is much larger when the load target is flushed.

Indeed, we observe the speculation window is much longer when the architectural load misses the cache, in which case we continue to observe LVP mispredictions up to 110 `mul` instructions. On the other hand, while the LVP still activates when the architectural load value is cached, we stop observing the stale load value being transmitted speculatively past 10 `mul`s. Given Apple documentation stating that each `mul` takes 3 cycles [105], our results translate to speculation windows of 330 (for cache misses) and 30 (for hits) cycles.

5.4.4 Inducing Memory Safety Violations

In this subsection, we investigate the security implications of LVP’s computation on stale load values, which we observed in subsection 5.4.3. Although the LVP does not predict arbitrary 64-bit values which could be pointers (cf. subsection 5.4.2), we show that the LVP is dangerous when coupled with layers of indirection. That is, incorrect load values can be used to perform out-of-bounds reads throughout the address space, and also to call functions that are never invoked in the program.

Reading Out of Bounds. As the LVP does not learn pointer values, we aim to use the load value as an index into an array of pointers. We introduce minor changes to the gadget function of Listing 5.13 from subsection 5.4.3, resulting in Listing 5.14.

Listing 5.14: Modified `gadget` function from Listing 5.13 that achieves 64-bit out-of-bounds reads. The LVP's stale load value causes the CPU to select the wrong pointer and dereference it.

```

1  uint8_t gadget(int offset, uint8_t *ptr) {
2      aop[foo] = ptr;
3      uint8_t val = *(mem + offset);
4      uint8_t *ptr = aop[val];
5      return *ptr;
6  }

```

In Line 1, we make `gadget` accept a pointer argument in addition to the offset into `mem`. Then, in Line 2, we insert this pointer into index `foo` of the array of pointers `aop`. In Line 3, we dereference `mem + offset` as before. Instead of returning the load value, we use it as the index into `aop` in Line 4, retrieving a pointer. Finally, in Line 5, we dereference the pointer and return the data at its address. Next, we show how the modified `gadget` function interacts with `aop` in Figure 5.12.

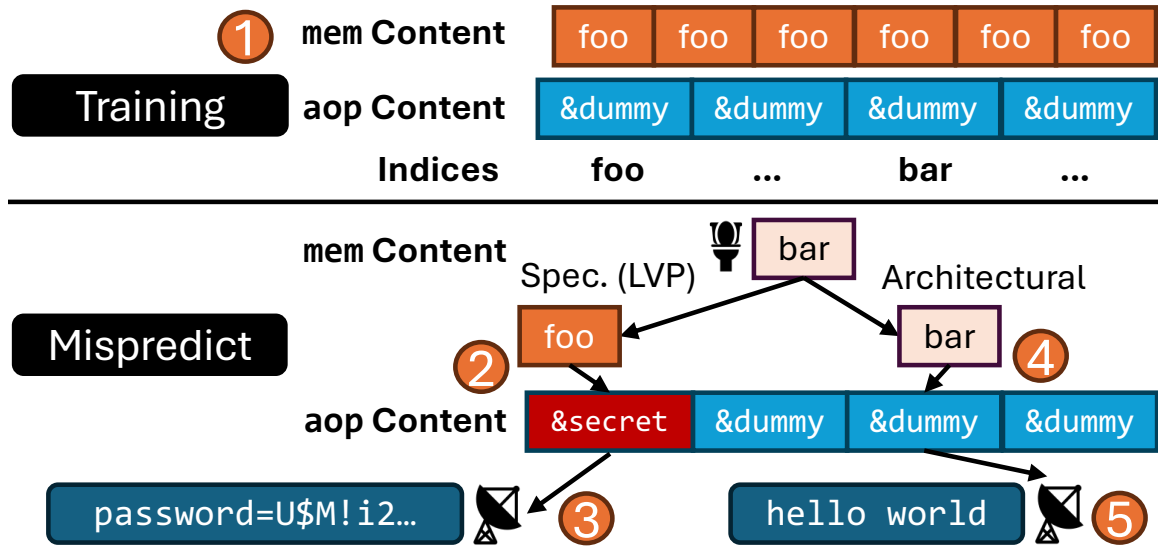


Figure 5.12: Graphical summary of our modified setup with indirection via an array of pointers to test for out-of-bounds reads during LVP speculation.

Training Phase. During LVP training (① in Figure 5.12), we fill `mem` with `foo`, and `aop` with the address of a dummy string. We run `gadget` with `ptr` set to `&dummy`, such that Line 2 does not change the contents of `aop[foo]`. Hence, when the LVP retrieves `foo` as the load value from `mem` in Line 3, the dummy string pointer is retrieved and dereferenced in Lines 4-5.

Misprediction Phase. Next, when we induce the misprediction in ②, we fill `mem` with `bar` and supply the address of `secret` to `gadget`, via `ptr`. Hence, `aop[foo]` now contains the `secret` pointer after Line 2. When the LVP activates in Line 3 because `mem` is flushed and assigns the stale load value `foo` to `val` instead of `bar`, this causes the `secret` pointer to be chosen and dereferenced in Lines 4-5. That is, `gadget` transiently returns data from the `secret` string, which then gets transmitted over Flush+Reload in ③. However, the CPU will eventually realize the correct load value is `bar` and replay the load into `mem`, as shown in ④. This results in the retrieval and transmission of the dummy string as well in ⑤, causing us to receive two values over Flush+Reload. Finally, since we know the dummy string’s content, we can simply filter out the value corresponding to it to recover `secret`.

Results. Using 250 training loads as in subsection 5.4.3, we perform the above setup to read the `secret` string for 100 trials. This results in mean and median accuracies of 0.97 and 1.00 (respectively) and throughput of 210,526 bits per second, demonstrating that the LVP can speculate into incorrect (and unsafe) data flows quite efficiently. Next, going beyond secrets in the current address space, we test for collisions in LVP state and LVP-induced memory safety violations across security boundaries in Appendix E. Our results indicate the LVP employs tagging using all instruction address bits and process ID, precluding its state from carrying over across processes. Moreover, after training the LVP in userspace, we observe that attempts to read kernel addresses are unsuccessful.

LVP in the macOS Kernel. One address space where out-of-bounds reads are particularly dangerous is that of the kernel. Thus, we ported our gadget from Listing 5.14 and

Figure 5.12 into macOS’s kernel space. We implement this as a kernel extension, wherein the training and misprediction phases can be invoked from a userspace driver program via `ioctl` syscalls. Remarkably, the LVP functions similarly in kernel-space: repeating the experiment to read a secret string (this time in kernel memory), we observe mean and median accuracies of 0.94 and 1.00 and throughput of 161,999 bits per second. While our scenario assumes the existence of suitable gadgets for LVP training in kernel space, we show that the a priori implications of in-kernel LVP activations are formidable.

Branching to Rogue Functions. We modify the out-of-bounds read experiment to determine if the LVP will also speculate into incorrect control flows. Firstly, we declare `aop` to hold function pointers instead of data pointers. Secondly, we replace Line 5 of Listing 5.14 with a call to the selected function pointer from `aop`, branching execution into the function’s entry point. Thirdly, we replace the secret and dummy strings with secret and dummy functions with the same function signature (arguments and return type), as we show in Figure 5.13. This diagram takes the place of the bottom half of Figure 5.12.

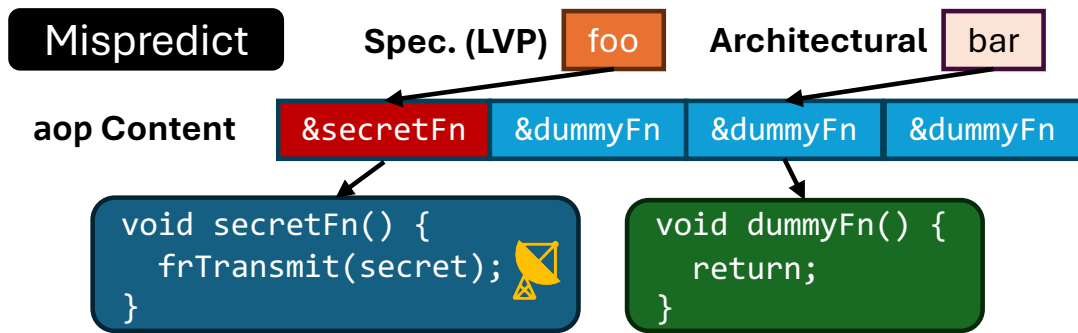


Figure 5.13: Modified lower half of Figure 5.12 to cause the stale load value `foo` to retrieve a function pointer and branch to it.

The secret function now contains the Flush+Reload transmission with a secret value. In contrast, the dummy function just returns. For our evaluation, we use 250 training loads and 100 trials as before, but measure how many times we correctly received the secret value and how many times the routine can be executed in one second. We receive the secret all 100 times and after a median of 0.000392 seconds: hence, our rogue function gadget can

be run about 2,551 times per second. Therefore, we conclude that the LVP is also effective at diverting control flow under speculation.

5.5 Attacking Safari with the LVP

In this section, we study the security implications of the LVP on a major component of the Apple ecosystem: the Safari web browser and its underlying browser engine, named WebKit. Since JavaScript is a weakly typed language, WebKit is responsible for checking types of variables under the hood in order to determine the operations it can perform on the variable. By causing the LVP to mispredict the type, we orchestrate FLOP-Data, an end-to-end attack capable of reading sensitive data from a cross-origin target webpage.

5.5.1 Value Prediction on Variable Types

We start our discussion of the LVP’s interaction with type checking in WebKit by introducing the typing mechanism. In WebKit, every JavaScript data structure starts with a header data structure named `JSCell`. Figure 5.14 shows its layout.

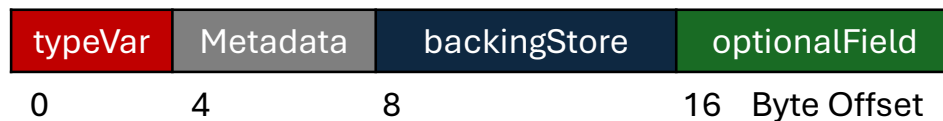


Figure 5.14: Memory layout of the `JSCell` header.

We note that the type information of the data structure is stored in the first 4 bytes of the `JSCell`. Therefore, for every object operation, WebKit starts executing the pseudocode in Listing 5.15, where it first performs a 4-byte load to the start of the `JSCell` to retrieve `typeVar` on Line 1.

Then, in Line 2, it compares the load value `typeVar` to the type that the code is expecting: if there is a mismatch, then the code aborts, throwing an exception in Line 3. That is, only after the type check passes will the code retrieve information from the rest of

Listing 5.15: Pseudocode for the type checking procedure for JavaScript data structures in WebKit. The highlighted load trains the LVP.

```
1 uint32_t inpType = input->typeVar;  
2 if (inpType != EXPECTED_TYPE)  
3     raiseException();  
4 doSomething(input->backingStore);  
5 doSomething(input->optionalField);
```

the `JSCell` such as `backingStore` and `optionalField`, as shown in Lines 4-5.

Considering LVP Activation Criteria. Now, we reexamine Line 1 of Listing 5.15 in the context of the LVP. As this is a 4-byte load, repeatedly running this code using an input of type `EXPECTED_TYPE` has the potential to train the LVP such that it predicts the load value will be `EXPECTED_TYPE`. That is, the edge case from subsection 5.4.2 where the LVP does not predict arbitrary values for 8-byte loads does not apply.

Moreover, we reflect on our findings from subsection 5.4.3: if we run Listing 5.15 again but with an input of a different type, the LVP may still use the (now stale) predicted load value for Line 1 and proceed to Line 2. The key insight is that the LVP's prediction will now cause the type check in Line 2 to incorrectly pass, resulting in the CPU proceeding to operate on the data structure (Lines 4-5). This paves the way for a speculative type confusion attack, which we now describe.

However, we also know from subsection 5.4.3 that the load in Line 1 must miss the cache during the misprediction run for a prolonged speculation window that can reach Lines 4-5. In contrast, when execution reaches those lines, `backingStore` and/or `optionalField` from Figure 5.14 must be cached for speculation to continue into the `doSomething` subroutines. This necessitates the `JSCell` header of the attacking data structure to be split across two cache lines.

Finding a JSCell Across Cache Lines. The `iLeakage` attack [98] demonstrated that `Intl.Locale` objects can be allocated such that the `typeVar` and `backingStore` variables from Figure 5.14 are on different cache lines. However, Apple has since in-

troduced a patch to WebKit [136] that ensures the two variables are always cache line-aligned, necessitating a novel attack vector. Accordingly, we turn our attention to the `optionalField` of the `JSCell`, which is only used by a small subset of JavaScript data structures compared to `backingStore`. It is this subset wherein we focus on typed arrays, a special class of arrays in JavaScript designed to handle raw binary data. We show their memory layout in Figure 5.15.

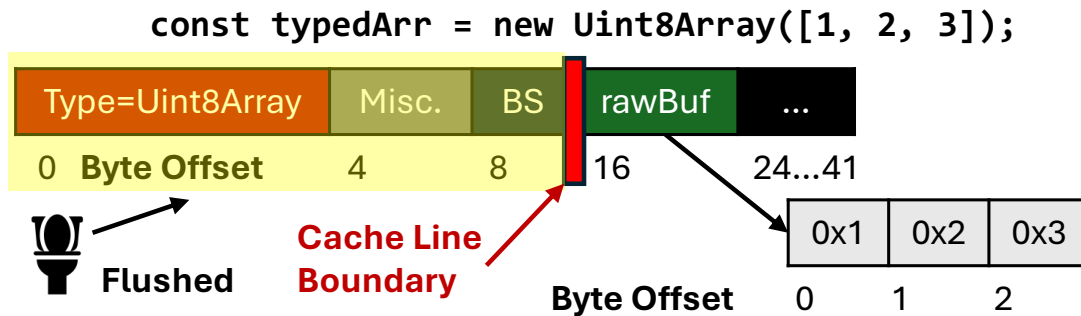


Figure 5.15: Memory layout of JavaScript typed arrays in WebKit. The `Uint8Array` (a class of typed array) declaration at the top of the figure produces this layout with a 3-byte buffer holding the data.

We observe that typed arrays leave the `backingStore` (denoted by `BS`) unused. Instead, the next variable (`rawBuf`) stores the address to their raw buffers. Furthermore, we observe that WebKit seldom allocates a typed array such that `Type`, `Misc.`, and `BS` are on one cache line, but `rawBuf` and some other metadata are on another cache line as shown in Figure 5.15. Thus, we can use typed arrays as the attacking data structure for speculative type confusion, since passing one to WebKit may activate the LVP when `Type` misses the cache.

5.5.2 Speculative Type Confusion

As an adversary, our end goal is to achieve a 64-bit read under speculation, which would allow us to recover secrets from anywhere in WebKit’s address space. In this subsection, we describe the exploit chain that begins from confusing a malicious typed array as another data structure and ends at retrieving data from a pointer which we control.

Target for Type Confusion. First, we would like the CPU to parse a typed array as something else due to the LVP mispredicting. However, this other data structure cannot use BS because BS for our malicious typed array would be flushed from the cache, preventing speculation from continuing. We find our candidate from an optimization used by WebKit: JavaScript objects that contain only a small number of member variables, or properties. While most objects use BS, WebKit stores small objects inline in memory, using fields after BS to hold the properties. We juxtapose the memory layouts of typed arrays and small objects in Figure 5.16.

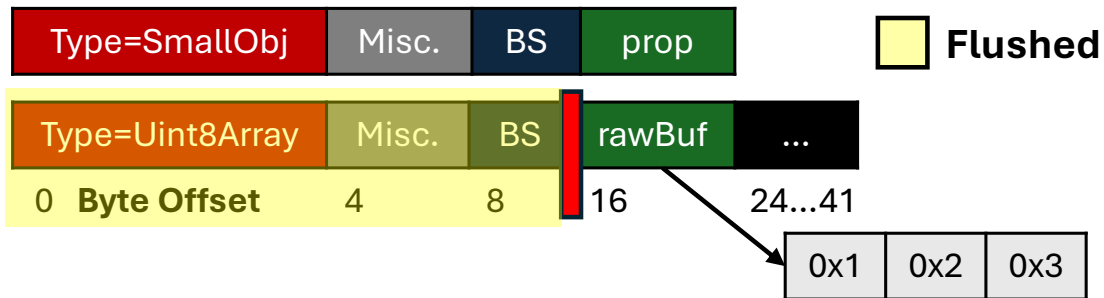


Figure 5.16: Memory layout of a small JavaScript object with an inline property (Top) and typed array split across cache lines (Bottom).

We observe that the attacker can modify the small object’s `prop` variable by changing the properties contained by that object. Furthermore, `prop` overlaps with `rawBuf` of the typed array in the memory layout. However, even though `prop` is 64 bits wide, we observe that the attacker cannot control all 64 bits of it due to WebKit’s sandboxing measures. That is, to prevent attacker-controlled JavaScript values from resembling pointers, WebKit poisons the values when storing them in memory such that they would never represent valid addresses. Therefore, a level of indirection is necessary.

Indirection with Strings. Consider the two JavaScript data structures presented in Figure 5.17. The top object `objWithStr` is a small object from Figure 5.16 (top), modified to contain a string. The bottom object `evil` is a typed array. Eventually, we would like the CPU to misinterpret `evil` as `objWithStr` under speculation, due to type confusion.

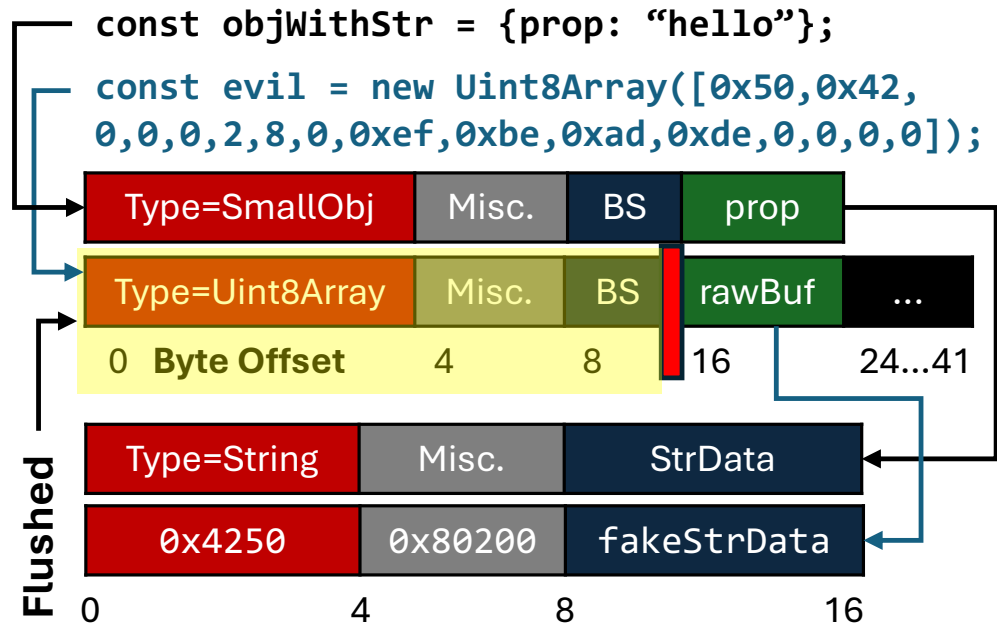


Figure 5.17: Memory layout of a small JavaScript object with a string (`objWithStr`) and typed array containing a malicious payload designed to imitate the string (`evil`). In this figure, the string’s backing store (pointed to by `StrData`) is not shown for simplicity. We note that the Apple M3 is a little-endian CPU, hence we write the raw bytes in reverse order.

We augment `objWithStr` with a string after observing that strings do not contain poisoned values. Furthermore, we observe that `rawBuf` of `evil` aligns in memory with the string’s metadata, which `prop` now points to. Akin to any other data structure, operations on strings are preceded by a type check. Therefore, for the CPU to operate on `rawBuf` under speculation, we must fill `rawBuf` with a payload to imitate the string’s metadata. Forging a data structure is usually a difficult task, as WebKit attempts to randomize the numerical values corresponding to each data structure’s type. However, as builtin types (e.g., strings) are allocated in a fixed order, we observe that the numeric type assigned to strings is always `0x4250`, and the miscellaneous data value in byte offsets 4-7 constantly equals (`0x80200`). We write these values into `evil`, causing `rawBuf` to resemble the first 8 bytes of a string.

Next, we focus on the `StrData` variable of the string object. This is a 64-bit pointer to the string’s underlying data structure, holding the string’s length and character buffer

(not shown for simplicity). Hence, by writing `fakeStrData` into `rawBuf`, it appears that we can create a fake 64-bit pointer to a string data structure. Despite this, we observe that WebKit does not expose the values from dereferencing `fakeStrData` to JavaScript code, as it instead parses the string’s data structure for fields such as `length`, `character buffer address`, etc. This precludes our code gadget from directly obtaining 64-bit reads via `fakeStrData`, requiring us to examine the string data structure that `StrData` typically points to.

JavaScript String Layout. We describe the full memory layout of a JavaScript `string` object in Figure 5.18.

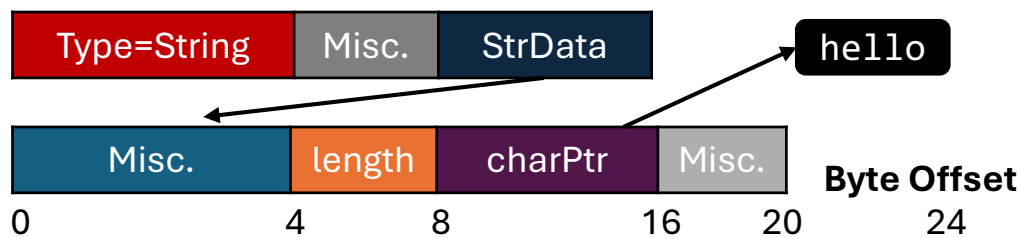


Figure 5.18: The full memory layout of the JavaScript string “hello”, which is the `.prop` property of the `objWithStr` object in Figure 5.17 above.

The underlying data includes the `length` and `charPtr`, which points to the first character. When indexing into a string from JavaScript, the index is checked against `length`. If it is in-bounds, the index is added to `charPtr`, and the CPU loads from the resulting address to retrieve a character to JavaScript code. Hence, if WebKit dereferences an attacker-controlled value instead of `charPtr`, this immediately leads to our end goal of 64-bit reads. To that aim, we assume there is a secret at memory address `addr` and there exists an attacker-controlled buffer. As the miscellaneous values in Figure 5.18 surrounding `length` and `charPtr` are constants, we can make this buffer resemble the string’s underlying data simply by writing these constants. Furthermore, we can write a large number for `length` such that any index would pass the check and write `addr` in place of `charPtr`. However, just as `StrData` points to this underlying data in Figure 5.18, we must write the address of this buffer in place of `fakeStrData` in Figure 5.17. As JavaScript does not

have pointers, we now face a challenge to sidestep these sandboxing measures.

Spraying Backing Stores on the Heap. We address this problem using, rather ironically, another JavaScript string. First, we make the insight that the string’s characters serve as an attacker-controlled contiguous buffer, where we can encode our payload from above. Next, strings can scale to massive lengths in WebKit, such as 1 GiB. Therefore, we repeat our payload to reach that length, effectively spraying multiple copies of the forged underlying data on the heap. Finally, we observe that declaring such a large string forces WebKit to allocate a new heap region. Remarkably, the new allocation consistently lands a couple bytes after $0x400000000$, making our spray technique extremely reliable. That is, we can replace `fakeStrData` in Figure 5.17’s `evil` with $0x400000100$ and expect this address to point to one of our sprayed instances. This completes our data structures for speculative type confusion, which we present in Figure 5.19.

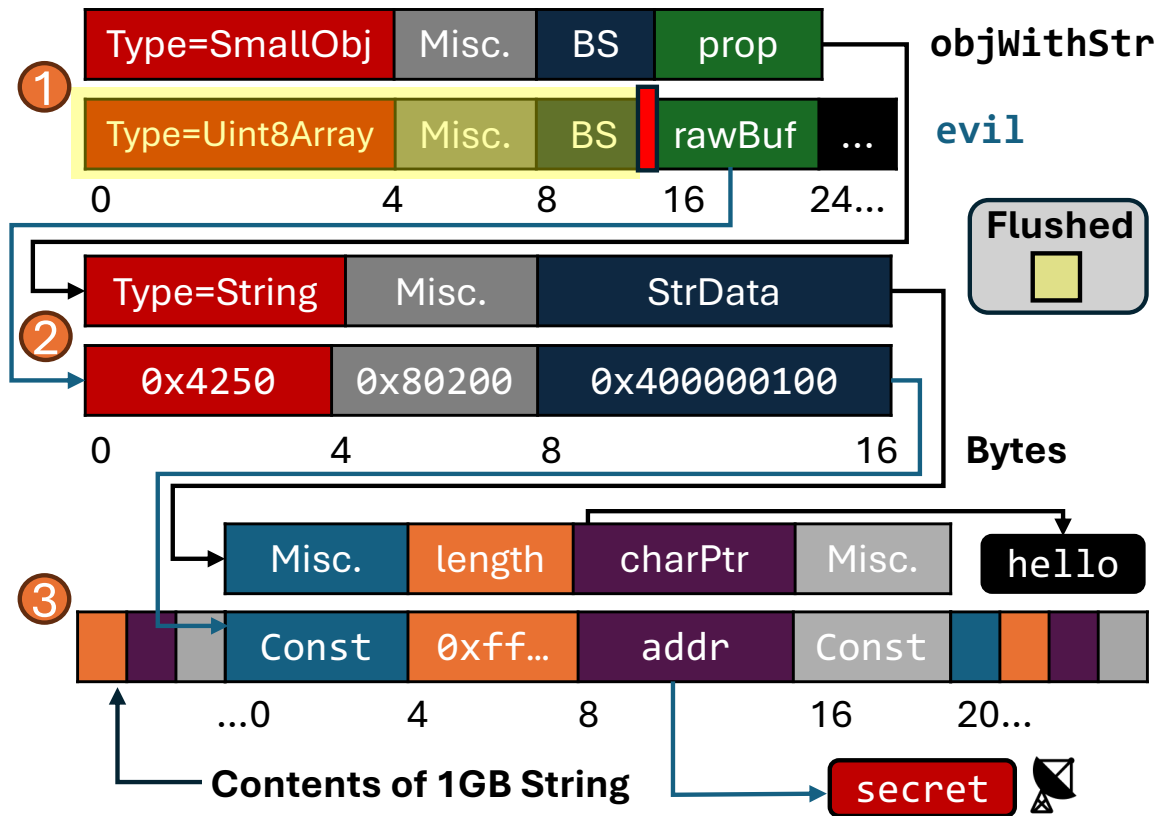


Figure 5.19: Our completed speculative type confusion primitive.

5.5.3 End-to-end Attack Overview

Using our data structures from Figure 5.19, we describe the remaining steps to orchestrate our attack end-to-end. First, we introduce our gadget in Listing 5.16.

Listing 5.16: Our gadget that operates on the data structures in Figure 5.19 to achieve transient 64-bit reads via speculative type confusion in Line 2.

```
1  function gadget(input, index) {  
2    const secret = input.prop.charCodeAt(index);  
3    channel.transmit(secret);  
4  }
```

Gadget Overview. We train the LVP on Listing 5.16 with `objWithStr`, whose layout is represented at the top of each pair of memory diagrams in Figure 5.19, as the `input` argument. In Line 2 (highlighted), we retrieve characters using the `charCodeAt` function of `objWithStr`'s string (i.e., its `.prop` property) several times with an in-bounds index, resulting in WebKit reading from `charPtr`. We then provide `evil` as `input` and an arbitrary `index` to Listing 5.16, located at the bottom of each pair in Figure 5.19. Then, we reason about how the CPU (and thus LVP) reacts to Line 2.

Type-confusing `evil` as `objWithStr`. ① of Figure 5.19 shows the speculative type confusion, where the LVP causes the CPU to operate on `evil` while thinking the `input` is `objWithStr` because `evil`'s type is flushed. As `evil`'s `rawBuf` is cached, the CPU continues to speculate and interprets `rawBuf` as `objWithStr`'s `prop` variable. Thus, execution proceeds to ②, where the CPU first checks if it is computing on a string such that it can eventually retrieve a character. The type check on `rawBuf` passes as `string`, since we have written `0x4250` and `0x80200` to `type` and `misc`. Subsequently, the CPU interprets `0x400000100` as a pointer (in place of `strData`) and proceeds by dereferencing it.

Achieving Transient 64-bit Reads. We now revisit the heap spray at ③, where one instance of our forged string data, encoded inside a 1 GiB-long JavaScript string, is located at `0x400000100`. The CPU, thinking this data structure is a string, attempts to retrieve

a character at `index`. As we have written the maximum possible value for `length`, we sidestep this check. Finally, the CPU interprets `addr` as `charPtr`, speculatively dereferencing the sum of `addr+index`, and returns the data to Line 2 under speculation. Finally, Line 3 leaks the data by encoding it in a microarchitectural covert channel. We describe the details of this covert channel and the full attack code implementation in Appendix F.

5.5.4 Evaluation

We first benchmark our 64-bit out-of-bounds read primitive, which we name FLOP-Data (a combination of the LVP and its ability to leak data). We host our attack code on a web server that is not publicly accessible. Then, we use FLOP-Data to read a buffer that we initialize with known data over 10 trials on a MacBook Pro with M3 CPU and 8 GB of RAM. Here, we observe a median accuracy of 89.58% and throughput of 0.492 bits per second. Subsequently, we evaluate the dangers of FLOP-Data on popular real-world websites. In these case studies, we assume the target user is authenticated into the target webpage via cookies stored by Safari.

Bringing Secrets Into the Address Space. Now that we can read from anywhere in the address space, we would like `addr` to point to meaningful targets, such as sensitive data from another webpage. As `addr` is a virtual address, we must cause the Safari process that renders the attacker’s webpage to also render the target webpage. Prior work [98] reported that the `window.open` API in JavaScript achieves this. Moreover, we observe we can combine `window.open` with the `onclick` event listener to open the target page without triggering pop-up blockers when the target user clicks anywhere on the attacker’s webpage. Hence, we cause clicks on our attacker webpage to call `window.open` with the target page’s URL as an argument, such that secrets in the target page become allocated in our address space.

Furthermore, we must ensure that `addr` probabilistically points to those secrets. For large allocations such as webpage DOMs, we observe that WebKit expands its heap at de-

terministic addresses (such as `0x400000000` from subsection 5.5.2) and that it enforces memory alignment of ≥ 16 bytes. This heavily reduces the entropy for a valid `addr`, oftentimes to brute-forceable 16-bit search spaces.

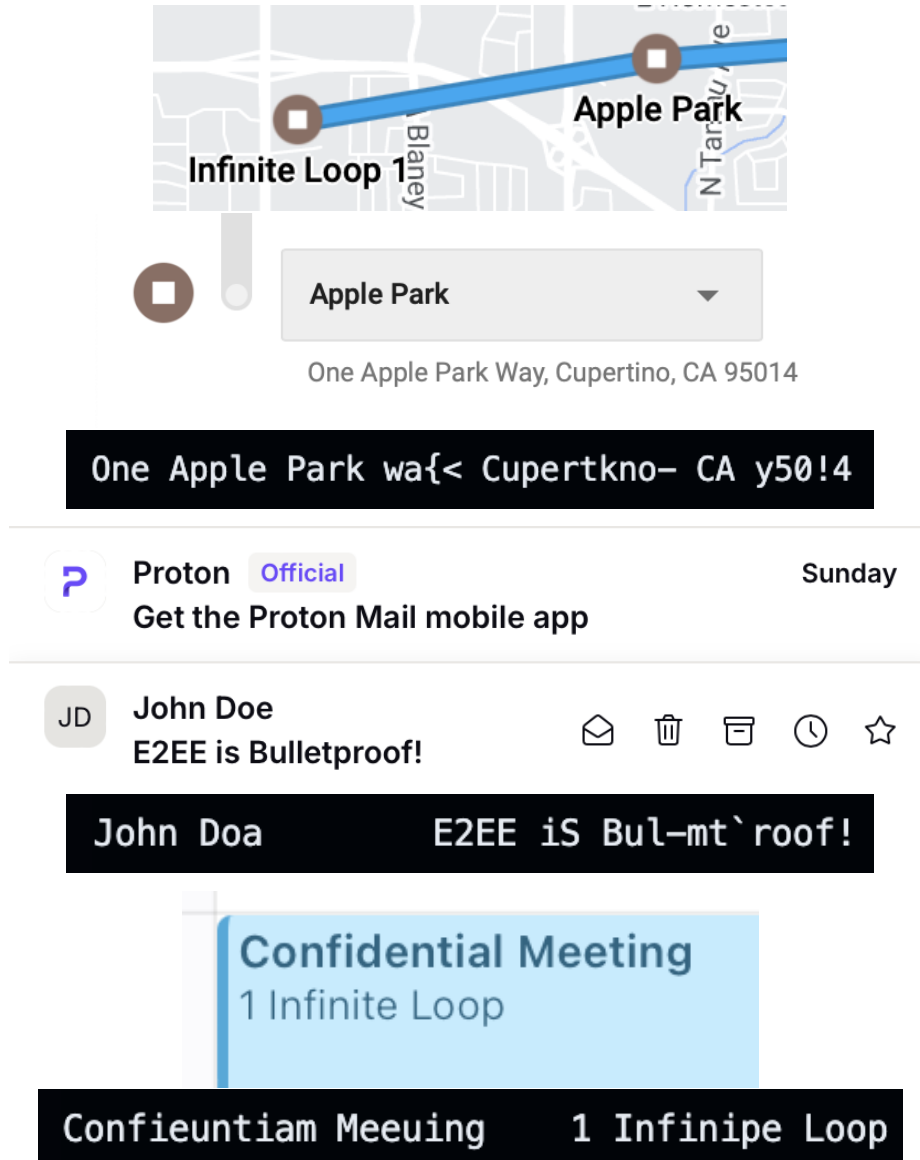


Figure 5.20: Sensitive data recovered with FLOP-Data from Google Maps Timeline (Top), Proton Mail’s inbox (Middle), and iCloud Calendar (Bottom).

Recovering Sensitive Data. We wrap up our evaluation by showing FLOP-Data’s ability to leak secrets from several web services. First, we notice that Google Maps’ Timeline page tracks the locations the account owner has visited by default, recovering an address in

Figure 5.20 (Top). Next, we recover the sender and subject of an email from the inbox of Proton Mail, an end-to-end encrypted email service, in Figure 5.20 (Middle). Finally, we head to Apple’s own iCloud Calendar, where we recover the name and location of a private event in Figure 5.20 (Bottom).

5.6 Attacking Google Chrome with the LVP

In this section, we show that the LVP not only disrupts data flow in practical end-to-end attacks, but also control flow beyond our experiment in subsection 5.4.4. We cause the CPU to execute the wrong function under speculation when running our WebAssembly gadget with Chrome, such that it treats a 64-bit integer as an address. Hence, we again achieve transient 64-bit reads to sensitive data throughout the address space.

5.6.1 WebAssembly Function Dispatch Table

First, we notice that Chrome prevents out-of-bounds reads for JavaScript objects by caging them in a 4 GiB memory region and limiting all pointers within this region to 32 bits. However, we observe that WebAssembly code is not subject to this caging restriction. Furthermore, we find that WebAssembly supports an interface where code can insert functions into a function dispatch table and subsequently invoke them by indexing into the table. We examine Chrome’s implementation of the function dispatch table in Figure 5.21.

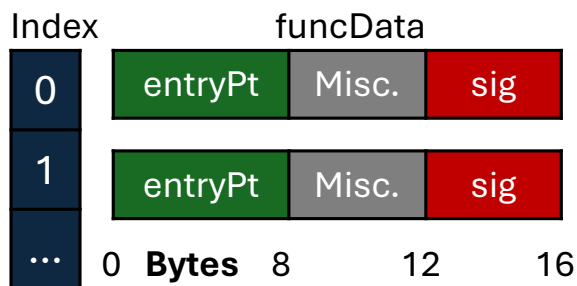


Figure 5.21: Memory layout of Chrome’s function dispatch table.

Dispatch Table Overview. In Figure 5.21, each table index has a data structure `funcData`. `funcData` is comprised of `entryPt`, a code pointer to the function’s entry

point, miscellaneous metadata, and `sig`, a 32-bit value representing the function’s signature (i.e., arguments and return type). Next, we show how functions are indirectly called in Listing 5.17.

Listing 5.17: Pseudocode for calling a function using the WebAssembly function dispatch table in Google Chrome. The highlighted loads train the LVP.

```
1 function call(args, int index) {  
2     funcData f = dispatchTable[index];  
3     uint32_t funcSig = f->sig;  
4     if (funcSig != EXPECTED_SIG)  
5         raiseException();  
6     setupArguments(args);  
7     f->entryPt();  
8 }
```

WebAssembly code can call functions from the dispatch table by specifying a 32-bit index in Line 1, and `args`, the arguments for the function. Line 2 indexes into the table, retrieving the `funcData` at the index. In Line 3, the CPU performs a 32-bit load to obtain `sig`. Line 4 compares the result to `EXPECTED_SIG`, a unique value encoding the function’s expected argument and return types. If there is a mismatch, Line 5 throws an exception to abort the code. Hence, Lines 4-5 safeguard the function at `entryPt` from being invoked with the wrong arguments. If this check passes, Line 6 sets up `args` and Line 7 branches to `entryPt` to execute the function.

5.6.2 Speculative Function Confusion

We recall our control-flow hijacking gadget from subsection 5.4.4. There, we used the LVP to mispredict a stale index into an array of function pointers, thus transiently invoking the wrong function. We note that the function dispatch table in Figure 5.21 is very similar, with it being an array of `funcData` rather than pointers. Also, in Listing 5.17, we observe `index` is 32 bits and therefore learnable by the LVP (cf. subsection 5.4.2).

Therefore, if `index` is not cached in Line 2, the LVP may activate on a stale prediction for `index`, leading the CPU to retrieve the wrong `funcData` under speculation. While

this may seem to achieve our goal of invoking the wrong `entryPt`, it also implies that the value of `sig` is incorrect compared to what the code expects (`EXPECTED_SIG`) in Line 4. Unfortunately, this causes control flow to head to the exception in Line 5 instead of the wrong `entryPt` in Line 7.

Bypassing Signature Checks. However, we observe that `sig`, like `index`, is 32 bits. Thus, we aim to redirect the control flow one more time by training the LVP to mispredict on both `index` and `sig`. The loads for `index` and `sig` are at different instruction addresses, hence the LVP does not mix training state between them (cf. subsection 5.4.2).

This requires `sig` not to be cached, such that the LVP will predict its value is `EXPECTED_SIG` and sidestep the argument check. However, if `entryPt` is also not cached, speculation will terminate in Line 7. Hence, we need to find an instance of `funcData` whose `entryPt` and `sig` are on different cache lines, and evict just the line containing `sig`. Fortunately, we discover such an instance by brute-forcing 32 table indices. This allows us to execute the wrong function on the wrong arguments, as we overview in Figure 5.22.

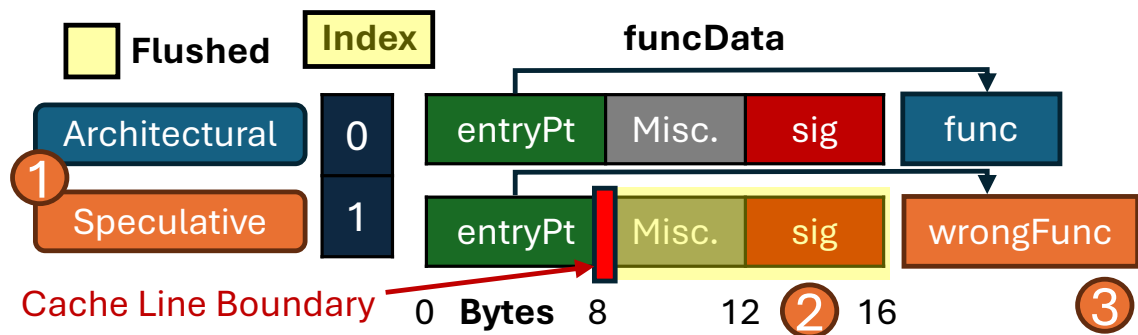


Figure 5.22: Our attack routine that causes two LVP mispredictions, disrupting control flow towards an incorrect function and also with the wrong arguments.

Hijacking Control Flow. As shown in Figure 5.22, we assume that the `funcData` split across cache lines is located at table index 1. We further assume that this `funcData`'s `sig` is not cached, as well as the `index` argument to Listing 5.17. Now, we invoke the `call` function with 0 for the `index` argument. In ① of Figure 5.22, the LVP first activates

on Line 2 due to the cache miss on `index`. The LVP mispredicts `index` to be 1, retrieving the `funcData` of `wrongFunc` instead of `func`.

In ②, the CPU runs Line 3 to retrieve the signature. However, since this is also a cache miss, the LVP activates again, mispredicting that the signature will be that of `func`. This circumvents speculation from heading to the exception in Line 5. Therefore, in Line 6, we cause the CPU to set up `func`'s arguments for `wrongFunc` instead. Then, since the wrong `entryPt` remains cached in Line 7, the CPU incorrectly diverts control flow to `wrongFunc` in ③.

Confusing Data as Addresses. With the ability to transiently call rogue functions with unchecked argument types, we investigate the arguments that WebAssembly functions can take. Due to sandboxing in the browser, WebAssembly does not support pointers. However, it does support references to valid `structs` in memory. Although the reference is not attacker-controllable, we observe that Chrome remarkably implements it as a 64-bit pointer. Hence, we use `struct` references in lieu of pointers, aiming to provide an attacker-controlled 64-bit integer to a WebAssembly function that expects a `struct` reference. To that aim, we introduce Listing 5.18.

Listing 5.18: Pseudocode for `func` and `wrongFunc` from Figure 5.22. We aim to run `wrongFunc` on a 64-bit integer, causing the CPU to treat it as a pointer.

```
1  struct readType { uint8_t data; }
2  function func(uint64_t arg) { return; }
3  function wrongFunc(struct readType* arg) {
4      uint8_t readVal = arg->data;
5      channel.transmit(readVal);
6  }
```

We first consider the `struct` in Line 1 that Chrome passes as a 64-bit pointer. Accessing the `data` variable dereferences that pointer, returning data at the `struct`'s address. Now, we consider the `func` and `wrongFunc` functions' implementation from Figure 5.22. We would like to run `wrongFunc` instead of `func` using `func`'s 64-bit integer argument, causing the CPU to confuse it with the 64-bit pointer to `struct`.

Function Implementations. `func` takes a 64-bit integer, but just returns, as shown in Line 2. In contrast, `wrongFunc` retrieves the data at the `struct` into `readVal` in Line 4, causing Chrome to perform a 64-bit pointer dereference. Then, in Line 5, `wrongFunc` uses a timer-resilient covert channel to transmit `readVal`, whose details we describe in Appendix G. Therefore, if we call `func` with a 64-bit integer but the CPU instead executes `wrongFunc` due to the LVP, the CPU will treat the integer as an address, and leak the data at that address through the cache. We name this primitive FLOP-Control, and describe the full implementation in Appendix H.

5.6.3 Evaluation

We first benchmark FLOP-Control by causing `addr` to point to a buffer we have initialized with known content. Next, we put Listing 5.18 on a non-publicly accessible web server and attempt to read the buffer for 10 trials using an M3 MacBook Pro with 8 GB of RAM. We report a median accuracy of 80.90% and throughput of 0.30 bits per second.

Locating Target Websites. Next, we would like `addr` to point to secrets from a target webpage. To do so, we must induce Chrome to render the target webpage in the attacker’s address space. However, Chrome enforces restrictions on which webpages can be co-rendered due to site isolation [90]. Here, the attacker and target webpages are allowed to share an address space only if their extended top-level domains (eTLD) and the prefix before the eTLD are identical, a rule known as eTLD+1. An eTLD can be a traditional top-level domain such as `.com` or `.org` in addition to approximately 15,000 domains on the Public Suffix List (PSL) [137], which specifies domains under which users can create their own sites, such as `github.io`. For this example, the eTLD+1 rule prevents one Chrome process from rendering both `attacker.github.io` and `target.github.io`, as well as `attacker.org` and `target.org`.

Therefore, we seek for webpages that satisfy three conditions. Firstly, the webpage must not be on the PSL, such that `attacker.site.tld` can share an address space with

`target.site.tld` (here, `site` is the common prefix). Secondly, the webpage must allow users to host their own JavaScript and WebAssembly on `attacker.site.tld`. Thirdly, `target.site.tld` must render secrets. Once we find such a target, we identify `addr` by running the attack in reverse: that is, we confuse addresses as data. First, as before, we locate a dispatch table entry that is split across cache lines. Then, by providing a 64-bit heap pointer to a function that transmits a 64-bit integer over our covert channel, we defeat ASLR from reading the pointer's value. This process takes approximately 30 seconds, constrained by our leakage throughput. Finally, breaking ASLR reduces the entropy for the secret's `addr` to 12 bits, allowing us to brute-force it.

Attacking Square. We find our target for FLOP-Control in the form of Square, a popular storefront service. The attacker can insert arbitrary JavaScript and WebAssembly into their storefront, which is hosted on `attacker.square.site`. Remarkably, Square is not on the PSL. This allows the attacker storefront to be co-rendered in Chrome with other storefront domains⁵ by calling `window.open` with their URLs, as demonstrated by prior work [31]. One such domain is the customer accounts page⁶, which shows the target user's saved credit card information and address if they are authenticated into the target storefront (see Figure 5.23 (Left)). As such, we recover the page's data in Figure 5.23 (Right).

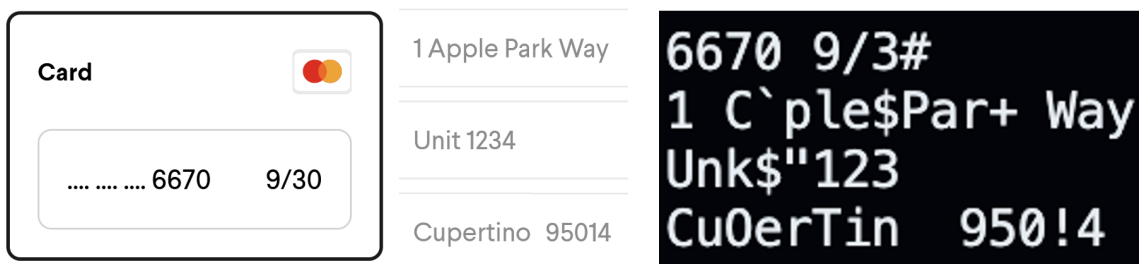


Figure 5.23: (Left) UI elements from Square's customer account page for a storefront. (Right) Recovered last 4 credit card number digits, expiration date, and billing address via FLOP-Control.

⁵Square storefronts are generally hosted at `storename.square.site`.

⁶For each storefront, the customer account page is located at `storename.square.site/s/customer-account`.

5.7 Mitigations

Now, we investigate how to mitigate the LVP-induced memory safety violations first introduced in subsection 5.4.4 and weaponized throughout section 5.5 and section 5.6. To that aim, we turn our attention to the Data Independent Timing (DIT) bit present in Armv8.4-A ISA and newer [117], and when set instructs the CPU that the latency to execute certain instructions should not correlate to the data in the operands.

Initially designed to protect constant-time code against dangerous microarchitectural optimizations [138], DIT is both readable and writable from unprivileged processes and is tracked across context switches [139]. Thus, changing the DIT bit for one process does not affect its value for other processes.

DIT Interactions with the LVP. With DIT already disabling data-dependent prefetching on the M3 [22], we now investigate if DIT also disables the LVP. We repeat the experiments of subsection 5.4.1, setting the DIT bit before running any instructions. On the M3 CPU's P-cores, we no longer observe a speedup on constant load values compared to random load values. Subsequently, we also repeat the misprediction experiment of subsection 5.4.3. Here, we do not receive the stale load value on the covert channel, regardless of how long we train the LVP. Therefore, we conclude that setting the DIT bit indeed disables the LVP. Hence, we recommend that developers patch their software to enable DIT on supported platforms, especially for code regions handling secrets or are untrusted.

For web browsers, this would correspond to setting the DIT bit when executing user-supplied JavaScript or WebAssembly, and on sensitive DOM operations such as password fields. As such, we patched Safari to set the DIT bit in the rendering process and observed that such a mitigation results in an overhead of 4.5% on the Speedometer 3.0 benchmark. In addition, we observe the overhead in native environments is even smaller, with 0.6% on average for our patched version of the BYTE Unix benchmark.

Mitigations for Browsers. In Safari, our cross-origin leaks of sensitive DOM data were

possible due to WebKit rendering the attacker and target webpages in the same address space. Safari lacks Site Isolation [90], which is implemented in Chrome and mandates process isolation for webpages from different origins. However, Site Isolation does not solve the co-rendering problem completely due to corner cases, as we observed with Square in subsection 5.6.3.

Next, we consider countermeasures for each browser engine. First, we suggest increasing the randomization entropy for types and memory allocations in Safari, as the lack thereof allowed us to create a fake string header and heap-spray the underlying string data⁷. Then for Chrome, we suggest caging the WebAssembly `structs` in a memory region, similarly to what it already does with JavaScript objects. This would allow references to them to be represented in base-relative offsets to that region (e.g., 32 bits) instead of 64-bit pointers, precluding an attacker from reading the whole address space⁸.

5.8 Conclusion and Future Work

In this chapter, we demonstrate that recent Apple CPUs perform load value prediction. After observing constant load values from a specific instruction address, the LVP bypasses subsequent loads, forwarding the same value to younger dependent instructions. This results in a speculation window that may compute arbitrarily on incorrect data. Using indirection, we show out-of-bounds addresses being read, the wrong function being executed, and finally sensitive login-protected data being exfiltrated across websites in Safari and Chrome. Finally, we identify that the LVP can be disabled via DIT.

On the other hand, we note that the DIT bit is not set (and thus the LVP is enabled) in the macOS kernel, where in subsection 5.4.4 we demonstrated the possibility of in-kernel LVP training and exploitation. However, as we were not able to train the LVP across userspace / kernel boundary, to successfully exploit the LVP in a kernel environment, an attacker must

⁷All types are stored in a sparse lookup table, and WebKit's memory allocation uses demand paging. Hence, we project the overhead to be negligible.

⁸The overhead is one addition, which is often just one CPU cycle.

find three types of gadgets: training gadgets containing loops of ≤ 32 -bit loads, converter gadgets causing the wrong load value to access a secret, and finally leak gadgets to exfiltrate the secret over covert channels. We leave the task of creating tools for finding such gadgets, as well as creating realistic LVP kernel exploits, to future work.

5.9 Ethics Considerations

We describe our adherence to ethics guidelines while evaluating our attacks in section 5.5 and section 5.6 and responsible disclosure.

Ethical Testing of Attacks. We own all devices used in our experiments, and these devices are free of any sensitive user data or personal information. These devices are only accessible to lab members, and are not exposed to unauthorized users. For online accounts, all of the accounts involved in our attacks are dummy accounts that we own, and are ensured not to contain any sensitive information. To reduce the risk to unrelated parties, we host the code for the attack on Safari (section 5.5) on a local web server that is not publicly accessible.

To overcome Site Isolation [90], the code of the attack on Chrome (section 5.6) has to be hosted in the same eTLD+1 domain as the target domain (cf. subsection 5.6.3), which in our case is a dummy Square storefront that does not sell anything. To protect unrelated users that inadvertently visit our storefront, we modify the attack code to check for a specific cookie value and disable the attack if the value is not found. Consequently, running the attack requires opening Chrome’s developer tools and manually adding a cookie, and thus we ensure it will not run on innocuous machines. Finally, we took down this storefront immediately after finishing our evaluation.

Responsible Disclosure. We disclosed our results to Apple’s Product Security Team on September 3, 2024 upon completing the initial version of the writeup. Apple has acknowledged our writeup, and after an internal investigation, communicated that they plan to address this in an upcoming security update without sharing further details.

CHAPTER 6

CONCLUSION

In this final chapter, we discuss the root causes that made the adversarial techniques possible throughout the works in this thesis. Furthermore, we propose measures for website administrators, browser vendors, and processor vendors to mitigate such root causes in the near future. Lastly, we suggest research directions for the side-channel security of web browsers that have yet to be explored.

Regarding Timer Degradation. As a fingerprinting and side-channel countermeasure, all major web browsers have degraded their timer resolution for JavaScript and WebAssembly code such that it is orders of magnitude coarser than what is needed to observe microarchitectural events. However, all works in this thesis demonstrate that it is not a panacea against browser side-channel attacks. Although degraded timers disincentivize such attacks by slowing down the throughput for recovering secrets perhaps from kilobytes per second to bits or dozens of bits per second, several microarchitectural timing amplification primitives show that data recovery is still possible. As such, we propound that more principled defenses are necessary, some of which we detail hereinafter.

Co-rendering of Webpages. Modern web browsers employ a multi-process architecture, consisting of a networking process that interacts with the operating system's TCP/IP stack, GPU process that interacts with the GPU driver for compositing, main process that handles the inter-process communication (IPC) among all other processes, and several renderer processes that handle the DOM and scripts of each webpage. As a countermeasure post-Spectre, each renderer process usually handles just one webpage. However, the works in this thesis show how to coerce the Chrome and Safari browsers' render processes to handle more than one webpage, which eventually leads to data being leaked across webpages (which is greatly facilitated when the target data is in the same address space). In this

chapter, we call this phenomenon co-rendering.

The attack surface for Safari is shown to be worse than that of Chrome, since arbitrary webpages can be co-rendered for the former, while only subdomains of the same webpage can be co-rendered for the latter. This is due to Safari’s lack of Site Isolation [90], which Chrome and Firefox have adopted. Throughout our disclosures, Apple has mentioned that Site Isolation for Safari is in development. However, due to the costs of overhauling Safari’s process architecture to support extensive IPC, the development timeline has stretched to over two years. We suggest maintaining Site Isolation as a high-priority task for Apple, and perhaps a short-term fix meanwhile. That is, chapter 4’s attack was possible due to the co-rendered websites also sharing memory heaps for JavaScript data structures, and therefore could be mitigated by heap isolation.

However, Site Isolation is also not a panacea for isolating cross-origin data across processes, since by default it allows subdomains to be co-rendered. This allows for corner cases where a web service developer could host untrusted user-uploaded scripts on one subdomain and sensitive login-protected information on another subdomain. Several corner cases with Bitbucket, Tumblr, and Google Photos were discovered in prior work [31] circa 2022, and chapter 5 demonstrates that the corner cases still exist in 2025 with Square. We strongly suggest webpages that are configured in the above layout to either isolate all user uploads to a separate domain name, or register their domain name on the Mozilla Foundation’s Public Suffix List [137] which disallows the co-rendering of subdomains.

Speculative Memory Safety. Moreover, it is imperative to raise awareness to the browser development community that speculative memory safety entails different threats from architectural memory safety. That is, bounds, length, and type checks to guard against (architectural) out-of-bounds accesses or type confusion are insufficient against speculation, as they can all be bypassed through mistraining. The works in this thesis demonstrate bypasses via a mistrained conditional branch, load address, and data value, but browser developers should anticipate future work to contribute more novel vectors for speculation.

Since Just-In-Time (JIT) compilation is entirely within the browser developer’s control, we suggest patching the JIT compiler to add memory fence instructions before security-critical checks since they terminate speculative execution.

Novel Microarchitectural Optimizations. Throughout chapter 4 and chapter 5, we observed that new microarchitectural primitives can open new attack surfaces. For both the Load Address Predictor (LAP) and Load Value Predictor (LVP) discovered in these works, it is notable that our experiments for crossing hardware-backed security domains had negative results. More specifically, we attempted to train each mechanism in an unprivileged user process, and then to activate the mechanism in either a kernel module or another user process. This results in two takeaways: isolating new microarchitectural primitives based on hardware-backed security domains is an excellent measure, and we suggest other processor vendors to test the same for both existing and new hardware optimizations. However, the attacks in this thesis breach web browser sandboxing in the same rendering process, which is a software-backed security domain.

This is where testing for hardware-backed security domains only falls short, and these corner cases are immensely more difficult to identify since the software-backed security domains are application-specific, as opposed to hardware-backed security domains being testable during the design verification process. Due to the LAP and LVP having been silently inserted into Apple’s next generation of processors without any press releases or documentation about them, finding their existence through reverse engineering experiments and reading patents or papers was a notably time-consuming task. We argue that vendor transparency about novel hardware mechanisms in their products’ technical reference manuals aids security researchers to identify these corner cases more rapidly.

Standards for Vulnerability Disclosure. Aside from the technical complications above, there exists an operational complication in the form of differing standards for disclosure ethics between communities in academia and industry. In the former, researchers are generally expected not to release live exploits or dangerously detailed proofs-of-concept until

a vendor has issued a patch, even if the vendor’s patch cycle is slow. However, in the latter, Google Project Zero has pioneered an approach to persuade vendors towards timely remediation by providing a strict 90-day embargo window until the vulnerabilities are publicly disclosed. Hence, we observe a tradeoff in long embargoes wherein users are protected in the short term, but tension exists when vendors appear to be dragging their feet in the face of urgent publication timelines. This indicates the necessity of a middle ground for protocol, such as vendors having to demonstrate meaningful patch progress periodically in order to justify an embargo extension.

6.1 Future Work

Total Address Space Isolation. With Chrome and Firefox having deployed Site Isolation and Safari developing it, the browser industry appears to be shifting towards isolating the data from each web origin into their own process. This elicits a philosophical question: if the corner cases of Site Isolation were to be resolved and there is total address space isolation among origins, is this a panacea for guarding cross-origin data from speculative and transient execution attacks?

On the topic of adversaries with native execution capabilities, one recent concerning advancement is the revelation of cross-hardware backed security domain attack surfaces with Spectre-v2, going across the user-kernel boundary, processes, and virtual machines [7, 8, 9, 10, 11, 13]. These attacks leverage fine-grained memory management primitives, such as placing branch instructions and branch targets at specific memory addresses to cause an index collision in the Branch Target Buffer (BTB) or Indirect Branch Target Buffer (IBTB), and then injecting the address of a disclosure gadget in the target’s address space respectively.

Hence, an interesting direction for future research would be to find how such collisions and value injections could be possible from browser-based code, given the lack of pointers and the inability to map targeted addresses with the `mmap` system call. One more question

is where to find not only disclosure gadgets but ones that are reachable (e.g., by a browser API call from the attacker webpage), and how to adapt existing gadget-searching tools for the Linux kernel (such as [8]) for web browser codebases. For example, a gadget in the networking process’s address space could leak `HttpOnly` cookies that are never exposed to scripts, but an attacker would need to know and convey both the memory address of the cookie data and covert channel parameters from their renderer process to the networking process – all in addition to finding and executing the gadget.

Native-to-Browser Attacks. We now reverse the previous idea of migrating native attack techniques to the browser (and their associated challenges). Side-channel attacks on web browsers have mostly assumed a remote and web-based attacker, as the threat model is much more lucrative than the native threat model of the attacker owning an unprivileged account on the same physical system. However, as a consequence, the side-channel attack surface of a native adversary on a browser target has much to be understood (outside of facilitating dictionary attacks on passwords, for instance [41]). Compared to the browser-based threat model, the attacker now gains pointers, fine-grained memory management, precise timing, and access to syscalls (which also opens more avenues for resource exhaustion). Hence, two questions follow: firstly, how can existing attacks be ‘supercharged’ by these additional capabilities? Moreover, what are the novel attack vectors that the browser industry should consider? Cross-process Spectre-v2, which we discussed before, may be a prime example for the latter question given that the attack is significantly easier to orchestrate from a native environment.

New APIs for Web Browsers. Web browsers have evolved over time from merely displaying simple formatted text and images to rendering games, handling online payments, and hosting conference calls. New APIs have added to their capabilities as users have shifted to living in the browser, and consequently have added to the size of the codebase that developers must audit and the range of attack surfaces that are missed in such audits. For instance, the inclusion of automatically executing untrusted JavaScript in a sandbox

was a major turning point, and then came tiered optimizing Just-In-Time (JIT) compilers for high-performance JavaScript web apps but with their own set of bugs (that were not present when scripts were simply interpreted line-by-line).

Fast-forwarding to the modern era, the inclusion of WebGPU in production builds of web browsers circa 2023-2025 has enabled graphics and parallel programming from browser environments. Hence, GPUs are now automatically executing untrusted code served from websites, leading to some preliminary works on fingerprinting via the new standard [140, 141]. Even more recently, WebLLM [142] has been introduced, which builds on top of WebGPU to run smaller-sized LLM inference tasks directly in the web browser. Google has also announced features in Chrome where browser-based code can interface with on-device models on recent hardware (that are also used by native applications) for tasks such as writing assistance and machine translation [143].

Hence, as security researchers, we must ask: what does all of this mean for attack surfaces in the browser? As even CSS has evolved to become powerful enough for fingerprinting attacks [144], scrutinizing these new APIs for security implications is an important direction for future research.

Appendices

APPENDIX A

LAP BEHAVIOR ON PAGE BOUNDARIES

Does the LAP Train Across Page Boundaries? We now seek to determine if the LAP maintains training state across page boundaries. To do so, we focus on two specific cases of the linked list workload from subsection 4.4.4, where all use a stride of 128 bytes, in Figure A.1.

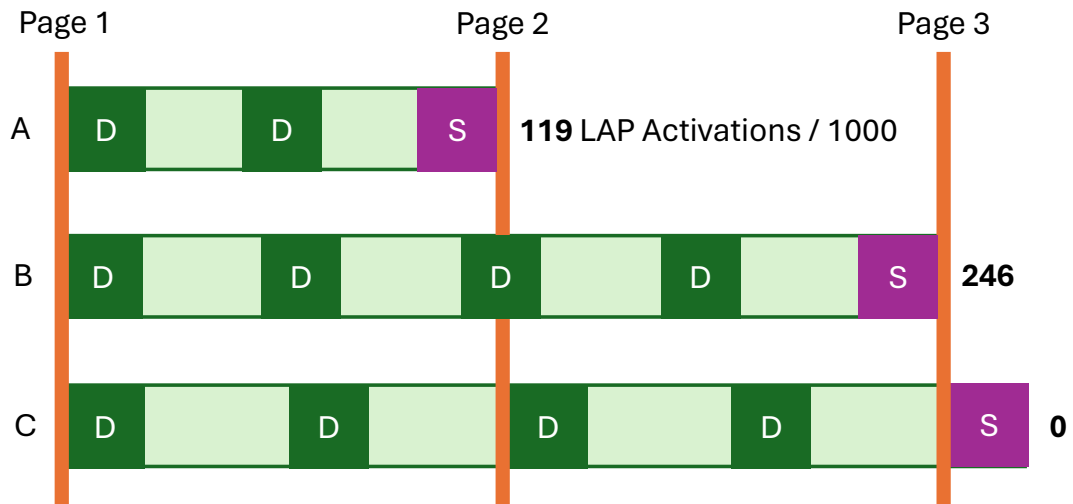


Figure A.1: Diagram of how the buffer used for training the LAP is allocated across pages for each case, along with the number of activations to the right. The ‘D’ and ‘S’ squares represent dummy and secret values, respectively.

The page size on Apple CPUs is 16 KiB. Here, Case ① trains the LAP on 128 nodes, such that training occurs across one page and the predicted load address is near the end of the page (offset $0 \times 3f80$). Case ② uses 256 nodes, training across two pages: likewise, the predicted load address is near the end of the second page. Hence, we compare the number of LAP activations across cases ① and ②. According to the heatmap in Figure 4.9, doubling the training length from 128 to 256 generally doubles the number of activations. As case ② has 128 training addresses on the first and second pages each, we expect to observe about double the activations if the LAP keeps state across pages, and the same

otherwise.

We present the number of activations for each case as the bolded number to the right of each memory diagram in Figure A.1. Comparing ① with ②, we answer the question in the affirmative from the approximately twofold increase coming from another page of training addresses.

Does the LAP Predict Across Page Boundaries? In addition, we determine if the LAP will generate a prediction on a new page. We start with Case ② from Figure A.1, and add Case ③ for comparison. Its 257 nodes (instead of 256) make training take place across two pages, but cause the predicted load address to be on a new page. Viewing the number of activations again, we answer the question in the negative this time, as we see no activations when the predicted address is on a new page.

APPENDIX B

CONFIRMING INSTRUCTION ADDRESS TAGGING

We use the training window experiment from subsection 4.4.5 as the groundwork, and the same training parameters (1,000 loads striding 32 bytes apart) to optimally activate the LAP. We now describe our modifications to first test for the presence of an instruction address tag, and then test whether the tag is a partial or full match.

Effects of Loop Unrolling. Firstly, we observe 997 LAP activations on Listing 4.8 with no `mul` instructions inserted. Then, we direct the compiler to completely unroll the for-loop in Line 2, and manually inspect the resulting binary. Here, we observe 0 LAP activations when the training loads are not from the same program counter. This observation indicates that the M2 LAP does indeed tag training states using some or all parts of the instruction address, and not the global history of load addresses.

The Control Experiment. Now, we aim to determine if the LAP uses a partial or full address match. To do so, we first design a control experiment where we divide training into two phases with a short interruption in between (which we know the LAP can tolerate from subsection 4.4.5) and ensure that we can still activate the LAP. During the interruption, we write a different secret value to the LAP’s predicted address. Our goal is to place the interruption where the LAP’s training state persists. That is, the second phase alone should not be able to activate the LAP. However, when we run both phases, we should receive the new secret value.

We start by putting Listing 4.8 in a function. We call the function to traverse the first $1000 - x$ linked list nodes for the first training phase. Then, we write the new secret value. Finally, we call the function again to traverse the last x nodes using the same instruction addresses. We increment x from 1 to 30 and measure the number of LAP activations (where we receive the new secret value) in Figure B.1 (Left). From the plot, we conclude

that $x = 30$ causes the LAP to reliably maintain training state.

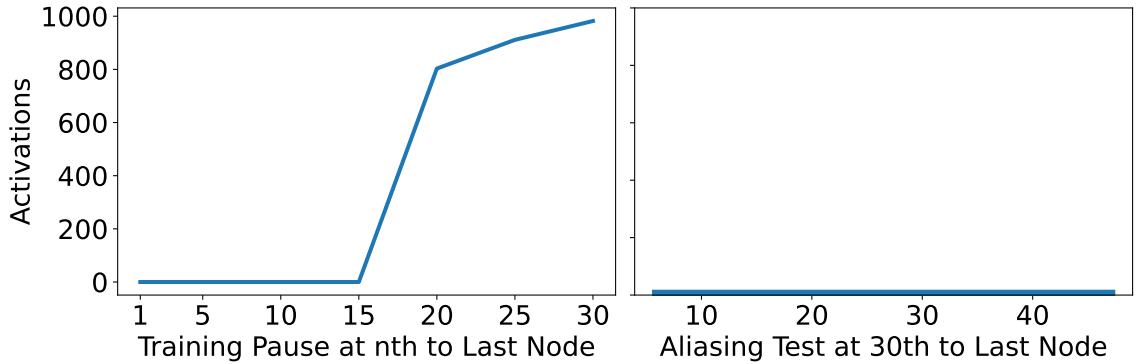


Figure B.1: (Left) Number of LAP activations when training is interrupted across a function call, starting at the last linked list node and ending at the 30th-to-last. (Right) Number of LAP activations when all but the last 30 nodes are traversed by the original function, and the rest are traversed by the aliased clone function.

Aliasing the Second Training Phase. Next, we create a clone of this function. We use linker scripts for `ld` to place the two functions apart at a fixed offset to alias the instruction addresses, ranging from the lowest 6 to 47 address bits. We traverse the first 970 nodes using the original function and write the new secret value, identically to the control experiment. However, we then traverse the remaining 30 nodes using the cloned function. With this setup, we observe 0 LAP activations for all aliasing offsets, as we show in Figure B.1 (Right). Therefore, we confirm that the LAP looks for a full match of all canonical address bits, since a partial match would lead to the LAP’s training state persisting (and thus the new secret being transmitted) at some point during the experiment.

APPENDIX C

TIMER-RESILIENT COVERT CHANNEL

In order to make cache hits distinguishable from misses in Safari, we reference the NOT gate-based cache amplification primitive from [77, Section 5], adjusting the speculation parameters for the M2 CPU. We run the amplifier 500 times when the target address is cached and 500 more times when it is evicted, in native and WebAssembly implementations. Table C.1 summarizes the timing distributions, with units in ms. We observe that they are clearly separable even in a web environment, allowing us to distinguish cache hits from misses with WebKit’s 1 ms timer.

Table C.1: Average runtime and standard deviation (both in milliseconds) for the NOT gate-based cache amplification primitive on native and WebAssembly runtimes on the Apple M2 CPU.

Test	Cached		Evicted	
	Avg. Time	Stdev.	Avg. Time	Stdev.
Native	5.12	0.30	9.40	1.14
WASM	7.54	0.51	11.87	1.17

APPENDIX D

MACOS DEVELOPMENT KERNEL SETUP

After installing the development kernel, we add `enable_skstb=1` to the boot arguments for the kernel to allow access to the `kern.sched.thread.bind_cpu` API in `sysctl` for setting core affinity. Then, we use a combination of kernel patches and extensions to allow user-space access to the `S3_2_c15_c0_0` system register (core cycles) and the `dc civac` (cache flush) instruction.

APPENDIX E

EXTENDED TESTS FOR ISOLATION

Having observed the LVP’s potential to cause memory safety violations when mispredicting in one address space, we now extend those experiments to determine which security boundaries the LVP can cross, and if we can cause collisions in the LVP’s internal state that can lead to training and exploitation happening in separate instructions or address spaces.

Testing for Instruction Address Aliasing. Earlier, in subsection 5.4.2, we have observed that the LVP trains with local scope, failing to activate (and cause speedups) when the training loop is unrolled. We now test this more thoroughly using the `gadget` function from subsection 5.4.3 and subsection 5.4.4. More specifically, we first clone the `gadget` function. Our goal is to train the LVP on the original `gadget`, then attempt to induce mispredictions when the cloned `gadget` executes.

To test for instruction address bits that are part of the page offset, we use the `C align` attribute to cause the least significant bits to alias. For more significant bits, we `mmap` code pages with the `MAP_FIXED` flag to map them at a specific page-aligned virtual address. Then, at the same page offset as the original `gadget`, we `memcpy` its instructions. These approaches allow us to alias from 8 to 46 least significant bits.

As a control, we train and evaluate the LVP only on the original `gadget` after implementing these changes, where we observe reliable mispredictions with 250 training loads. However, when evaluating on the cloned `gadget`, we no longer observe them regardless of the number of aliased bits. Hence, we conclude that the LVP likely tags its internal state with the full instruction address, precluding aliasing attacks.

Testing for Process Isolation and ASID Collisions. Now, we aim to run the training and misprediction routines in different processes. To that aim, we map the `mem` buffer with the `MAP_SHARED` flag such that writes to `mem` from one process become visible to

the other. We also call `fork` just before training, using the child process to train the LVP. Subsequently, the parent runs the misprediction routine.

In a variant of this experiment, we keep forking until the parent and child processes share the same Address Space Identifier (ASID). An AArch64 feature, ASIDs are assigned by the kernel and are used to tag page table base registers among other microarchitectural features. ASIDs are either 8 or 16 bits wide, with the M3 implementing the former. Given that the process limit for unprivileged users in macOS is 2,666, it is straightforward to spawn at most 256 child processes and achieve two processes with different Process Identifier (PID)s but the same ASID.

Again, as controls, we both train and evaluate the LVP in only the parent or child process and achieve reliable mispredictions. Nonetheless, we do not observe any signal from mispredictions when the two steps are performed in different processes even when the ASIDs collide, indicating that the LVP may employ PID-tagging in addition to PC-tagging.

Reading Kernel Addresses from Userspace. Finally, we augment subsection 5.4.4's experiment with a kernel extension that initializes a secret string in kernel memory and discloses only its address. We then write that address into `aop[foo]` before causing the LVP to mispredict to determine if LVP activations from userspace instructions can also read kernel memory. However, we do not receive any data over the covert channel this time, possibly indicating that speculation may terminate when loading from a kernel address.

APPENDIX F

FLOP-DATA IMPLEMENTATION DETAILS

We present the full attack pseudocode that operates on Figure 5.19’s `objWithStr` and `evil` in Listing F.19.

Listing F.19: Pseudocode in JavaScript for our speculative type confusion attack. The `||` operator in Lines 9 and 13 represents concatenation.

```
1  function gadget(input, index) {
2      const secret = input.prop.charCodeAt(index);
3      channel.transmit(secret);
4  }
5  // Data structure setup
6  const objWithStr = {prop: "hello"};
7  const fakeStr = 0x4250 || 0x80200 || 0x400000100;
8  let evil = undefined;
9  for (let i = 0; i < 13; i++)
10     evil = new Uint8Array(fakeStr);
11  const spray = CONST || 232-1 || addr || CONST;
12  const str1GB = spray.repeat(SPRAY_SIZE);
13  // Training phase
14  for (let i = 0; i < TRAIN_REPS; i++)
15     gadget(objWithStr, 0);
16  // Attack phase
17  partialEvict(evil);
18  if (false) gadget(evil, leakIndex);
19  return channel.receive();
```

Covert Channel Details. Once `addr` is transiently dereferenced after Line 2 and the data at `addr` is returned to the `secret` variable, we must consider that WebKit restricts the timer precision in JavaScript to 1 ms to deter fingerprinting and side-channel attacks [48]. This necessitates a cache amplification primitive for us to recover the secret using a cache covert channel. To that aim, we adapt a technique pioneered by Google’s `leaky.page` attack [29] that amplifies one L1 data cache hit or miss into tens of thousands by leveraging the replacement policy. This amplified covert channel is represented as the `channel` variable in Line 3.

Data Structure Setup. Before invoking `gadget`, we initialize data structures for the attack. We declare `objWithStr` on Line 6 of Listing F.19, and craft our payload for the typed array’s data on Line 7. Then, in Lines 8-10, we declare 13 typed arrays with this payload, but discard all but the last declaration. Empirically, we observe that the 13th allocation usually creates a typed array instance split across cache lines for a newly spawned WebKit rendering process, and thus we assign this one to the `evil` variable. Next, Lines 11-12 spray the heap with the 1 GiB string that contains repeats of our second payload, which resembles the underlying data of a string. Line 11 forms the payload by sandwiching the largest possible value for `length`, followed by `addr`, between the miscellaneous constants. Line 12 constructs a new string that repeats Line 11 until the result is 1 GiB long.

Training Phase. Recalling from subsection 5.4.3 that 250 training loads suffice to train the LVP reliably, we run the for-loop in Lines 14-15 with the `TRAIN_REPS` variable set to 250. Within the loop in Line 15, we repeatedly invoke `gadget` with 0 as the index such that it is in bounds. Also, we give `objWithStr` as the input, aiming to mistrain the LVP that the load value for `Type` should be `SmallObj`.

Attack Phase. Subsequently, in Line 17, we evict the first 16 bytes of `evil` from the cache to force the LVP into speculation, while keeping the rest cached for speculation to continue to `addr`. But now, we reason about WebKit’s behavior after transmitting `secret`. Eventually, the CPU will realize the LVP’s prediction was incorrect and replay execution from ① of Figure 5.19. Because we provided `evil`, a typed array, to code that was expecting `objWithStr`, WebKit will raise an exception that prevents us from using our attack more than once. To avoid this, we enclose the call to `gadget` on a mispredicted if-statement in Line 18 for speculative hiding [2]. This causes the exception to be raised under speculation, making the incorrect execution invisible to Safari. Finally, Line 19 recovers the secret via the amplified covert channel.

APPENDIX G

CACHE COVERT CHANNEL DETAILS

As a side-channel countermeasure, the timer resolution in Chrome is restricted to 100 μ s [47]. Hence, to recover secrets encoded in the cache state, we require the ability to distinguish cache hits from misses even when the timer is orders of magnitude coarser than the memory access latency. To that aim, we adapt the timing amplification primitive from [77] which uses speculation windows to cascade one cache hit or miss into thousands of hits or misses, allowing us to distinguish the two outcomes even with 100 μ s resolution.

APPENDIX H

FLOP-CONTROL IMPLEMENTATION DETAILS

We present the full attack pseudocode in Listing H.20, which is the extended version of Listing 5.18.

Listing H.20: Pseudocode for our control flow attack on Chrome, where a mispredicted table index and signature cause a 64-bit number to be treated as a pointer. `dispatchTable.call` in the highlighted lines runs Listing 5.17.

```
1  struct readType { uint8_t data; }
2  function func(uint64_t arg) { return; }
3  function wrongFunc(struct readType* arg) {
4      uint8_t readVal = arg->data;
5      channel.transmit(readVal);
6  }
7  // Training phase
8  uint64_t addr = &secret;
9  dispatchTable[1].set(func);
10 for (let i = 0; i < TRAIN_REPS; i++)
11     dispatchTable.call(addr, 1);
12 // Attack phase
13 dispatchTable[0].set(func);
14 dispatchTable[1].set(wrongFunc);
15 int idx = 0;
16 evict(idx, dispatchTable[1].sig);
17 dispatchTable.call(addr, idx);
18 return channel.receive();
```

Training Phase. We start by training the LVP on the table index and function signature. In Line 8, we assume there is a secret in our address space, but store its address as a 64-bit integer `addr`. In Line 9, we put `func` in table index 1 for LVP training, and we assume the set function updates `entryPt` and `sig`. Then, in Lines 10-11, we call `func` through the dispatch table, which causes Listing 5.17 to run. Hence, we train the LVP to predict 1 for the table index, and `func`'s `sig` value for the signature. We set the `TRAIN_REPS` parameter in Line 10 to 250, since we observed in subsection 5.4.3 that 250 training loads suffice for reliable mispredictions.

Attack Phase. Next, we reassign `func` to table index 0 in Line 13. In Line 14, we cause index 1 to refer to `wrongFunc` now. Then, in Line 15, we set a zeroed variable `idx` to be passed as the dispatch table index. As shown in Figure 5.22, we now assume that `funcData` at index 1 can be partially evicted. Thus, in Line 16, we evict both `idx` and `sig` while keeping the entry point to `wrongFunc` cached. We call the dispatch table again in Line 17, where `func` should be called because `idx` is 0. However, `idx` is not cached, causing the LVP to mispredict the table index as 1 (① of Figure 5.22).

Under speculation, we sidestep the signature check from the second LVP misprediction, due to `wrongFunc`'s `sig` also not being cached. Therefore, the CPU regards `addr` as a valid argument, diverting control flow into `wrongFunc` along with `addr`. Then, `wrongFunc` treats `addr` as a `struct` reference, loading the secret at `addr` and leaking it through the cache. Subsequently, the CPU will realize it mispredicted the table index and execute Listing 5.17 architecturally, calling `func`. However, `func` just returns since it was provided a valid argument. Finally, the CPU runs Line 18, allowing us to retrieve the secret over the covert channel.

REFERENCES

- [1] P. Kocher *et al.*, “Spectre attacks: Exploiting speculative execution,” *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.
- [2] M. Lipp *et al.*, “Meltdown: Reading kernel memory from user space,” in *USENIX Security*, 2018.
- [3] V. Kiriansky and C. Waldspurger, “Speculative buffer overflows: Attacks and defenses,” *arXiv preprint arXiv:1807.03757*, 2018.
- [4] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, “NetSpectre: Read Arbitrary Memory over Network,” in *ESORICS’19*, 2019.
- [5] A. Bhattacharyya *et al.*, “SMoTherSpectre: Exploiting speculative execution through port contention,” in *CCS*, 2019.
- [6] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, “Speculative probing: Hacking blind in the Spectre era,” in *CCS*, 2020.
- [7] J. Wikner and K. Razavi, “Breaking the barrier: Post-barrier spectre attacks,” in *2025 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2025, pp. 3516–3533.
- [8] S. Wiebing, A. de Faveri Tron, H. Bos, and C. Giuffrida, “Inspectre gadget: Inspecting the residual attack surface of cross-privilege spectre v2,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 577–594.
- [9] S. Wiebing and C. Giuffrida, “Training solo: On the limitations of domain isolation against spectre-v2 attacks,” in *2025 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2025, pp. 3599–3616.
- [10] J.-C. Graf, S. Rügge, A. Hajiabadi, and K. Razavi, “Vmscape: Exposing and exploiting incomplete branch predictor isolation in cloud environments,” in *2026 IEEE Symposium on Security and Privacy (SP)*, 2026.
- [11] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, “Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks,” in *USENIX Security*, 2022.
- [12] L. Li, H. Yavarzadeh, and D. Tullsen, “Indirector: {high-precision} branch target injection attacks exploiting the indirect branch predictor,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 2137–2154.

- [13] S. Rügge, J. Wikner, and K. Razavi, “Branch privilege injection: Compromising spectre v2 hardware mitigations by exploiting branch predictor race conditions,” in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 2615–2631.
- [14] G. Maisuradze and C. Rossow, “Ret2spec: Speculative execution using return stack buffers,” in *CCS*, 2018.
- [15] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, “Spectre returns! speculation attacks using the return stack buffer,” in *WOOT*, 2018.
- [16] J. Wikner and K. Razavi, “RETBLEED: Arbitrary speculative code execution with return instructions,” in *USENIX Security*, 2022.
- [17] J. Horn, *Speculative execution, variant 4: Speculative store bypass*, <https://bugs.chromium.org/p/project-zero/issues/detail?id=1528>, 2018.
- [18] C. Liu *et al.*, “Uncovering and exploiting amd speculative memory access predictors for fun and profit,” in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2024, pp. 31–45.
- [19] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, “Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks,” in *USENIX Security*, 2021.
- [20] M. Andryscio, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, “On subnormal floating point and abnormal timing,” in *IEEE SP*, 2015.
- [21] J. R. S. Vicarte *et al.*, “Augury: Using data memory-dependent prefetchers to leak data at rest,” in *IEEE SP*, 2022.
- [22] B. Chen *et al.*, “Gofetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers,” in *USENIX Security*, 2024.
- [23] A. Wang *et al.*, “Peek-a-walk: Leaking secrets via page walk side channels,” in *2025 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2025, pp. 3534–3548.
- [24] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, “The spy in the sandbox: Practical cache attacks in JavaScript and their implications,” in *CCS*, 2015.
- [25] P. Vila, B. Köpf, and J. F. Morales, “Theory and practice of finding eviction sets,” in *IEEE SP*, 2019.

- [26] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, “Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript,” in *Financial Cryptography and Data Security*, 2017.
- [27] A. Shusterman *et al.*, “Robust website fingerprinting through the cache occupancy channel,” in *USENIX Security*, 2019.
- [28] A. Shusterman, A. Agarwal, S. O’Connell, D. Genkin, Y. Oren, and Y. Yarom, “Prime + Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses,” in *USENIX Security*, 2021.
- [29] Google, *Spectre*, <https://leaky.page>, 2021.
- [30] J. Wikner, C. Giuffrida, H. Bos, and K. Razavi, “Spring: Spectre returning in the browser with speculative load queuing and deep stacks,” in *WOOT*, 2022.
- [31] A. Agarwal *et al.*, “Spook.js: Attacking chrome strict site isolation via speculative execution,” in *IEEE SP*, 2022.
- [32] P. Stone, “Pixel perfect timing attacks with HTML5,” *Context Information Security (White Paper)*, 2013.
- [33] H. Taneja, J. Kim, J. J. Xu, S. van Schaik, D. Genkin, and Y. Yarom, “Hot pixels: Frequency, power, and temperature attacks on gpu and arm socs,” in *USENIX Security*, 2023.
- [34] Y. Wang *et al.*, “Gpu. zip: On the side-channel implications of hardware-based graphical data compression,” in *2024 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2024, pp. 3716–3734.
- [35] S. O’Connell *et al.*, “Pixel thief: Exploiting {svg} filter leakage in firefox and chrome,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 3331–3348.
- [36] T. Rokicki, C. Maurice, M. Botvinnik, and Y. Oren, “Port contention goes portable: Port contention side channels in web browsers,” in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 1182–1194.
- [37] D. Gruss, C. Maurice, and S. Mangard, “Rowhammer.js: A remote software-induced fault attack in JavaScript,” in *DIMVA*, 2016.
- [38] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, “SMASH: Synchronized many-sided Rowhammer attacks from JavaScript,” in *USENIX Security*, 2021.

- [39] I. Kang *et al.*, “Sledgehammer: Amplifying rowhammer via bank-level parallelism,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 1597–1614.
- [40] F. De Ridder, P. Jattke, and K. Razavi, “Posthammer: Pervasive browser-based rowhammer attacks with postponed refresh commands,” in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 5661–5678.
- [41] A. Kwong *et al.*, “Checking passwords on leaky computers: A side channel analysis of chrome’s password leak detect protocol,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7107–7124.
- [42] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, “Drive-by key-extraction cache attacks from portable code,” in *ACNS*, 2018.
- [43] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, “Grand pwning unit: Accelerating microarchitectural attacks with the GPU,” in *IEEE SP*, 2018.
- [44] D. Kohlbrenner and H. Shacham, “On the effectiveness of mitigations against floating-point timing channels,” in *USENIX Security*, 2017.
- [45] D. Kohlbrenner and H. Shacham, “Trusted browsers for uncertain times,” in *USENIX Security*, 2016.
- [46] R. McIlroy, J. Sevcík, T. Tebbi, B. L. Titzer, and T. Verwaest, *Spectre is here to stay: An analysis of side-channels and speculative execution*, CoRR abs/1902.05178, 2019.
- [47] Y. Weiss and E. Kitamura, *Aligning timers with cross origin isolation restrictions*, <https://developer.chrome.com/blog/cross-origin-isolated-hr-timers/>, 2021.
- [48] R. Niwa, *Reduce the precision of ”high” resolution time to 1ms*, <https://github.com/WebKit/WebKit/commit/25e575313d12e97a9e6c2b1d9b6ddd1d510e01a9>, 2018.
- [49] MDN Contributors, *Performance.now() - web APIs - MDN*, https://developer.mozilla.org/en-US/docs/Web/API/Performance/now#reduced_time_precision, 2022.
- [50] A. Deveria, *Shared array buffer — can i use... support tables for html5, css3, etc*, <https://caniuse.com/sharedarraybuffer>, 2022.
- [51] H. Martin, *M1ssing register access controls leak ELO state*, <https://m1racles.com/>, 2021.

- [52] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, “PACMAN: Attacking Arm pointer authentication with speculative execution,” in *ISCA*, 2022.
- [53] D. Gullasch, E. Bangerter, and S. Krenn, “Cache games—bringing access-based cache attacks on AES to practice,” in *IEEE SP*, 2011.
- [54] Y. Yarom and K. Falkner, “Flush+Reload: A high resolution, low noise, L3 cache side-channel attack,” in *USENIX Security*, 2014.
- [55] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, “Flush+Flush: A fast and stealthy cache attack,” in *DIMVA*, 2016.
- [56] C. Percival, “Cache missing for fun and profit,” in *BSDCan*, 2005.
- [57] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-level cache side-channel attacks are practical,” in *IEEE SP*, 2015.
- [58] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, “A high-resolution side-channel attack on last-level cache,” in *DAC*, 2016.
- [59] G. Irazoqui, T. Eisenbarth, and B. Sunar, “S\$A: A shared cache attack that works across cores and defies VM sandboxing—and its application to AES,” in *IEEE SP*, 2015.
- [60] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, “Prime + Abort: A timer-free high-precision L3 cache attack using Intel TSX,” in *USENIX Security*, 2017.
- [61] S. van Schaik, C. Giuffrida, H. Bos, and K. Razavi, “Malicious Management Unit: Why stopping cache attacks in software is harder than you think,” in *USENIX Security*, 2018.
- [62] A. Purnal, F. Turan, and I. Verbauwhede, “Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks,” in *CCS*, 2021.
- [63] J. van Bulck *et al.*, “Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution,” in *USENIX Security*, 2018.
- [64] O. Weisse *et al.*, *Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution*, <https://foreshadowattack.eu/foreshadow-NG.pdf>, 2018.
- [65] S. van Schaik *et al.*, “Rogue in-flight data load,” in *IEEE SP*, 2019.
- [66] C. Canella *et al.*, “Fallout: Leaking data on Meltdown-resistant CPUs,” in *CCS*, 2019.

- [67] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, “CacheOut: Leaking data on Intel CPUs via cache evictions,” in *USENIX Security*, 2021.
- [68] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, *SGAxe: How SGX fails in practice*, <https://sgaxe.com/files/SGAxe.pdf>, 2020.
- [69] J. Stecklina and T. Prescher, “LazyFP: Leaking FPU register state using microarchitectural side-channels,” *arXiv preprint arXiv:1806.07480*, 2018.
- [70] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, “Crosstalk: Speculative data leaks across cores are real,” in *IEEE SP*, 2021.
- [71] C. Canella *et al.*, “A systematic evaluation of transient execution attacks and defenses,” in *USENIX Security*, 2019.
- [72] A. Lutas and D. Lutas, *Security implications of speculatively executing segmentation related instructions on Intel CPUs*, <https://businessresources.bitdefender.com/hubfs/noindex/Bitdefender-WhitePaper-INTEL-CPU.pdf>, 2019.
- [73] J. van Bulck *et al.*, “LVI: Hijacking transient execution through microarchitectural load value injection,” in *IEEE SP*, 2020.
- [74] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, “Take a way: Exploring the security implications of AMD’s cache way predictors,” in *Asia CCS*, 2020.
- [75] D. Evtuyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, “Branchscope: A new side-channel attack on directional branch predictor,” in *ASPLOS*, 2018.
- [76] S. Islam *et al.*, “SPOILER: Speculative load hazards boost Rowhammer and cache attacks,” in *USENIX Security*, 2019.
- [77] D. Katzman, W. Kosasih, C. Chuengsatiansup, E. Ronen, and Y. Yarom, “The gates of time: Improving cache attacks with transient execution,” in *USENIX Security*, 2023.
- [78] L. Hetterich and M. Schwarz, “Branch different-spectre attacks on apple silicon,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2022, pp. 116–135.
- [79] Apple, *Xnu/osfmk/arm/kpc_arm.c*, https://github.com/apple-oss-distributions/xnu/blob/e6231be02a03711ca404e5121a151b24afbff733/osfmk/arm/kpc_arm.c, 2021.

- [80] D. Lemire, *Counting cycles and instructions on the Apple M1 processor*, <https://lemire.me/blog/2021/03/24/counting-cycles-and-instructions-on-the-apple-m1-processor/>, 2021.
- [81] E. Kitamura and D. Denicola, *Why you need "cross-origin isolated" for powerful features*, <https://web.dev/why-coop-coep/>, 2021.
- [82] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU.," in *NDSS*, 2017.
- [83] H. Xiao and S. Ainsworth, "Hacky racers: Exploiting instruction-level parallelism to generate stealthy fine-grained timers," *arXiv preprint arXiv:2211.14647*, 2022.
- [84] A. Purnal, M. Bognar, F. Piessens, and I. Verbauwhede, "Showtime: Amplifying arbitrary cpu timing side channels," in *ACM AsiaCCS 2023*, 2023.
- [85] Intel, *Intel transactional synchronization extensions (intel tsx) memory and performance monitoring update for intel processors*, <https://www.intel.com/content/www/us/en/support/articles/000059422/processors.html>, 2021.
- [86] R. Zhang, T. Kim, D. Weber, and M. Schwarz, "(m) wait for it: Bridging the gap between microarchitectural and architectural side channels," in *USENIX Security*, 2023.
- [87] G. Chen *et al.*, "Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races," in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 178–194.
- [88] G. Haas, S. Potluri, and A. Aysu, "Itimed: Cache attacks on the apple a10 fusion soc," in *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, IEEE, 2021, pp. 80–90.
- [89] D. Johnson, *Apple M1 microarchitecture research*, <https://dougallj.github.io/applecpu/firestorm-int.html>, 2021.
- [90] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: Process separation for web sites within the browser," in *USENIX Security*, 2019.
- [91] Mozilla, *Project Fission*, https://wiki.mozilla.org/Project_Fission, 2021.
- [92] J. Wilander, *Intelligent tracking prevention*, <https://webkit.org/blog/7675/intelligent-tracking-prevention/>, 2017.
- [93] N. Hadad and J. Afek, *Overcoming (some) Spectre browser mitigations*, <https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/>, 2018.

- [94] O. Kirzner and A. Morrison, “An analysis of speculative type confusion vulnerabilities in the wild,” in *USENIX Security*, 2021.
- [95] IEEE Standard Board, “IEEE standard for floating-point arithmetic,” IEEE, IEEE Std 754-2008, 2008.
- [96] Apple, *Process swap on cross-site window.open behind a flag*, <https://github.com/WebKit/WebKit/pull/10169>, 2023.
- [97] Apple, *About speculative execution vulnerabilities in arm-based and intel cpus*, <https://support.apple.com/en-us/HT208394>, 2018.
- [98] J. Kim, S. van Schaik, D. Genkin, and Y. Yarom, “iLeakage: browser-based timerless speculative execution attacks on apple devices,” in *ACM CCS*, 2023.
- [99] A. Perais and A. Sez nec, “Practical data value speculation for future high-end processors,” in *IEEE HPCA*, 2014.
- [100] A. Perais and A. Sez nec, “Bebop: A cost effective predictor infrastructure for superscalar value prediction,” in *IEEE HPCA*, 2015.
- [101] R. Sheikh, H. W. Cain, and R. Damodaran, “Load value prediction via path-based address prediction: Avoiding mispredictions due to conflicting stores,” in *IEEE/ACM MICRO*, Cambridge, Massachusetts, 2017.
- [102] R. Sheikh and D. Hower, “Efficient load value prediction using multiple predictors and filters,” in *IEEE HPCA*, 2019.
- [103] H. Wang, M. Ibrahim, S. Mittal, and A. Jog, “Address-stride assisted approximate load value prediction in gpus,” in *ACM ICS*, Phoenix, Arizona, 2019.
- [104] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, “Value locality and load value prediction,” in *ACM ASPLOS*, 1996.
- [105] Apple, *Apple silicon cpu optimization guide: 3.0*, https://developer.apple.com/documentation/apple-silicon/cpu-optimization-guide?changes=_9, 2024.
- [106] R. M. Tomasulo, “An efficient algorithm for exploiting multiple arithmetic units,” *IBM Journal of research and Development*, vol. 11, no. 1, pp. 25–33, 1967.
- [107] Y. C. Chou, V. Gautam, W.-H. Lien, K. N. Kothari, and M. Agarwal, *Early load execution via constant address and stride prediction*, US Patent 11,829,763, Nov. 2023.

- [108] H. S. Barowski and R. Hilgendorf, *Universal load address/value prediction using stride-based pattern history and last-value prediction in a two-level table scheme*, US Patent 6,986,027, Jan. 2006.
- [109] B. K. Parady, *Threshold-based load address prediction and new thread identification in a multithreaded microprocessor*, US Patent 6,907,520, Jun. 2005.
- [110] Apple, *Xnu/osfmk/vm/vm_map.c at main*, https://github.com/apple-oss-distributions/xnu/blob/94d3b452840153a99b38a3a9659680b2a006908e/osfmk/vm/vm_map.c, 2024.
- [111] Apple, *Xnu/bsd/kern/mach_loader.c at main*, https://github.com/apple-oss-distributions/xnu/blob/main/bsd/kern/mach_loader.c, 2024.
- [112] Apple, *Xnu/osfmk/arm64/proc_reg.h at main*, https://github.com/apple-oss-distributions/xnu/blob/main/osfmk/arm64/proc_reg.h, 2024.
- [113] Apple, *Xnu/external_headers/mach-o/loader.h at main*, https://github.com/apple-oss-distributions/xnu/blob/94d3b452840153a99b38a3a9659680b2a006908e/EXTERNAL_HEADERS/mach-o/loader.h, 2024.
- [114] F. Pizlo, *All about libpas, phil's super fast malloc*, <https://github.com/WebKit/WebKit/blob/main/Source/bmalloc/libpas/Documentation.md>, 2024.
- [115] Apple, *Internals - webkit documentation*, <https://docs.webkit.org/Deep20Dive/Libpas/Internals.html>, 2024.
- [116] Apple, *Energy efficiency guide for mac apps: Prioritize work at the task level*, https://developer.apple.com/library/archive/documentation/Performance/Conceptual/power_efficiency_guidelines_osx/PrioritizeWorkAtTheTaskLevel.html, 2016.
- [117] ARM, *How is instruction timing affected by the feat_dit architectural feature?* <https://developer.arm.com/documentation/ka005181/latest/>, 2024.
- [118] H. Martin, *Found the dmp disable chicken bit*, <https://social.treehouse.systems/@marcan/112238385679496096>, 2024.
- [119] Asahi, *Hw:arm system registers*, <https://github.com/AsahiLinux/docs/wiki/HW3AARM-System-Registers>, 2023.
- [120] PassMark, *Passmark - intel core i5-2500t @ 2.30ghz*, <https://www.cpubenchmark.net/cpu.php>, 2024.
- [121] Apple, *Webkit contributor meeting 2022*, <https://docs.webkit.org/Other/Contributor20Meetings/ContributorMeeting2022.html>, 2022.

- [122] L.-A. Daniel, M. Bognar, J. Noorman, S. Bardin, T. Rezk, and F. Piessens, “ProSpeCT: Provably secure speculation for the constant-time policy,” in *USENIX Security*, 2023.
- [123] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, “Invisispec: Making speculative execution invisible in the cache hierarchy,” in *IEEE/ACM MICRO*, 2018.
- [124] M. Taram, A. Venkat, and D. Tullsen, “Context-sensitive fencing: Securing speculative execution via microcode customization,” in *ACM ASPLOS*, 2019.
- [125] J. Yu, A. Dutta, T. Jaeger, D. Kohlbrenner, and C. W. Fletcher, “Synchronization storage channels (s2c): Timer-less cache Side-Channel attacks on the apple m1 via hardware synchronization instructions,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.
- [126] Intel, *Fast store forwarding predictor*, <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/fast-store-forwarding-predictor.html>, 2022.
- [127] AMD, *Security analysis of AMD predictive store forwarding*, <https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf>, 2021.
- [128] K. Kalaitzidis and A. Sez nec, “Leveraging value equality prediction for value speculation,” *ACM Transactions on Architecture and Code Optimization (TACO)*, 2020.
- [129] D. Moghimi, “Downfall: Exploiting speculative data gathering,” in *USENIX Security*, 2023.
- [130] F. Gabbay and A. Mendelson, “Using value prediction to increase the power of speculative execution hardware,” *ACM Transactions on Computer Systems (TOCS)*, 1998.
- [131] A. Perais and A. Sez nec, “EOLE: Combining static and dynamic scheduling through value prediction to reduce complexity and increase performance,” *ACM Transactions on Computer Systems (TOCS)*, 2016.
- [132] S. Bandishte, J. Gaur, Z. Sperber, L. Rappoport, A. Yoaz, and S. Subramoney, “Focused value prediction,” in *ACM/IEEE ISCA*, 2020.
- [133] L. Orosa, R. Azevedo, and O. Mutlu, “Avpp: Address-first value-next predictor with value prefetching for improving the efficiency of load value prediction,” *ACM Transactions on Architecture and Code Optimization (TACO)*, 2018.

- [134] G. Anders, *Exploring Mach-O, part 3*, <https://gpanders.com/blog/exploring-mach-o-part-3/>, 2022.
- [135] A. Shilov, *Apple M3 CPUs hit 4.05 GHz, challenge Raptor Lake in Geekbench*, <https://www.tomshardware.com/pc-components/cpus/apple-m3-cpus-hit-405-ghz-challenge-raptor-lake-in-geekbench>, 2023.
- [136] M. Lam, *Adjust precise allocation alignment offset to also factor in cache line alignment requirements*, <https://github.com/WebKit/WebKit/commit/842bf586330dbf74f9e2d09d50c818ca3f792988>, 2023.
- [137] Mozilla Foundation, *Public suffix list*, <https://publicsuffix.org/>, 2020.
- [138] Apple, *Writing ARM64 code for Apple platforms*, <https://developer.apple.com/documentation/xcode/writing-arm64-code-for-apple-platforms>, 2024.
- [139] A. Holdings, *Processor state in exception handling*, <https://developer.arm.com/documentation/100933/0100/Processor-state-in-exception-handling>, 2024.
- [140] E. Ferguson, A. Wilson, and H. Naghibijouybari, “Webgpu-spy: Finding fingerprints in the sandbox through gpu cache attacks,” in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 158–171.
- [141] L. Giner *et al.*, “Generic and automated drive-by gpu cache attacks from the browser,” in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 128–140.
- [142] C. F. Ruan *et al.*, “Webllm: A high-performance in-browser llm inference engine,” *arXiv preprint arXiv:2412.15803*, 2024.
- [143] A. Klepper, *Ai apis are in stable and origin trials, with new early preview program apis*, <https://developer.chrome.com/blog/ai-api-updates-io25>, 2025.
- [144] L. Trampert, D. Weber, L. Gerlach, C. Rossow, and M. Schwarz, “Cascading spy sheets: Exploiting the complexity of modern css for email and browser fingerprinting,” in *Network and Distributed System Security Symposium (NDSS)*, 2025.