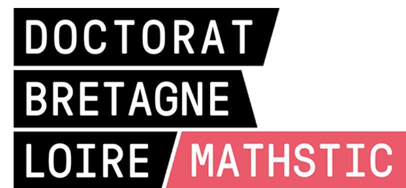


THÈSE DE DOCTORAT DE

L'INSTITUT NATIONAL DES
SCIENCES APPLIQUÉES RENNES

ÉCOLE DOCTORALE Nº 601
Mathématiques et Sciences et Technologies
de l'Information et de la Communication
Spécialité : *Informatique*

Par
Thomas ROKICKI

Side Channels in Web Browsers : Applications to Security and Privacy

Thèse présentée et soutenue à IRISA, Rennes, le 29 Novembre 2022
Unité de recherche : UMR 6074

Rapporteurs avant soutenance :

Jan REINEKE	Professeur, Saarland University
Billy Bob BRUMLEY	Maître de Conférence, Tampere University

Composition du Jury :

Président :	Lilian BOSSUET	Professeur, Université Jean Monnet
Examinateur·rice·s :	Jan REINEKE	Professeur, Saarland University
	Billy Bob BRUMLEY	Maître de Conférence, Tampere University
	Veelasha MOONSAMY	Chargée de Recherche, Ruhr University Bochum
	Walter RUDAMETKIN	Professeur, Université de Rennes 1
Dir. de thèse :	Gildas AVOINE	Professeur, INSA Rennes, CNRS, IRISA, France
Encadrante de thèse :	Clémentine MAURICE	Chargée de Recherche, Univ Lille, CNRS, Inria , France

Thèse de doctorat de Thomas ROKICKI

Thomas ROKICKI

Supervisor: Clémentine MAURICE

Co-supervisor: Gildas AVOINE

29/11/2022

Résumé en Français

Les processeurs modernes sont des objets technologiques remarquablement optimisés. La vitesse de traitement étant le principal argument de vente des microprocesseurs, les fabricants se sont attachés à augmenter les performances des processeurs lors de leur développement, chaque année proposant de nouvelles générations dotées de meilleures caractéristiques que les précédentes.

Comme les progrès en matière de taille et de performances des semi-conducteurs ont ralenti dans l'ensemble de l'industrie au cours des dernières décennies, le simple fait d'ajouter des transistors à une puce est devenu insuffisant pour continuer l'augmentation régulière des performances. En réaction, les fabricants de processeurs se sont concentrés sur les optimisations microarchitecturales pour améliorer davantage les performances de génération en génération. Ces optimisations visent à augmenter la vitesse de traitement en proposant plus de parallélisme et en réduisant les temps de latence du matériel. Les processeurs multicœurs, les caches, ou l'exécution spéculative sont quelques exemples d'optimisations microarchitecturales. Chaque génération apporte des modifications à la microarchitecture, avec de plus en plus de caches ou d'unités d'exécutions dans un pipeline complexe. Par conception, ces optimisations visent à apporter plus de performance en accélérant l'exécution. Dans cette course à la performance, la sécurité de la microarchitecture tend à être négligée. En particulier, les effets secondaires de ces optimisations sur l'execution des calculs peuvent entraîner des fuites d'informations sur les valeurs secrètes utilisées par le système dans ce qu'on appelle des *canaux auxiliaires.*

En 1996, Paul Kocher [Koc96] a révélé que l'accélération dynamique apportée par les caches introduit des différences de temps d'exécution dans les implémentations à temps constant des algorithmes cryptographiques. Un utilisateur malveillant peut utiliser ces différences de temps microscopiques pour extraire des secrets telles que des clés privées. Cette découverte de la première attaque par minutage a ouvert la voie à un champ de recherche plus large sur les canaux auxiliaires.

Les canaux auxiliaires physiques ont fait l'objet de nombreuses recherches. Ils exploitent les effets secondaires physiques du processeur pendant le calcul pour en extraire les secrets des calculs délicats. Le premier canal auxiliaire physique exploitait la consommation d'énergie de la puce [KJJ99] pour attaquer les implémentations cryptographiques, mais de nombreuses sources de fuites physiques ont été étudiées, notamment les émissions acoustiques [ST04, GST14], les émissions électromagnétiques [RR01], les émissions optiques [SNK+12] ou encore la température [HS13]. Cependant, sauf exception notable [LKO+21], ces attaques nécessitent un accès physique au dispositif cible afin de mesurer les effets secondaires du calcul.

Par opposition, les *canaux auxiliaires microarchitecturaux* sont purement logiciels. Ils exploitent les optimisations basées sur les données des processeurs modernes pour déduire des informations privées. Depuis la première attaque temporelle de Kocher [Koc96], les caches

ont été la source la plus étudiée de canaux auxiliaires microarchitecturaux, mais la recherche a prouvé que d'autres composants peuvent également être exploités pour monter des attaques par canal auxiliaire. Au cours de la dernière décennie, les chercheurs ont ciblés de nombreux composants microarchitecturaux, en ont compris le fonctionnement, et ont utilisé leur comportement pour en extraire des secrets. Les canaux auxiliaires ont été exploités, entre autres, à partir des prédicteurs de branchement [AKS07], de la DRAM [PGM+16], ou de la traduction d'adresses [GRBG18]. Les canaux auxiliaires microarchitecturaux ont été initialement mis en place pour faire fuir les données des implémentations cryptographiques, mais leur portée s'est considérablement élargie, avec notamment des attaques contre l'ASLR [HWH13], la communication entre machines virtuelles [MWS+17], la détection des frappes au clavier [GSM15] et la surveillance des utilisateurs [SKH+19]. Plus inquiétant encore, les canaux auxiliaires microarchitecturaux constituent une brique essentielle d'une nouvelle classe d'attaques microarchitecturales : les attaques par exécution transiente. Introduites en 2018 avec Spectre [KHF+19] et Meltdown [LSG+20], ces attaques utilisent les effets secondaires des instructions transientes, i.e., des instructions exécutées mais jamais engagées dans l'architecture, pour faire fuir des données à travers les barrières de sécurité logicielles et matérielles.

Les canaux auxiliaires microarchitecturaux exploitent le partage des composants matériels entre l'attaquant et la victime pour divulguer des informations. Par conséquent, ils ont un prérequis commun : *l'attaquant doit exécuter du code sur la machine de la victime.* Cette condition préalable essentielle restreint considérablement la surface d'attaque des canaux auxiliaires microarchitecturaux, car les utilisateurs se méfient de plus en plus des logiciels inconnus et ont tendance à ne télécharger des logiciels qu'à partir de sources fiables. Le risque de télécharger des logiciels malicieux est même réduit sur les plateformes mobiles, où les applications sont téléchargées à partir d'un magasin d'applications contrôlé par un éditeur, où chaque morceau de code est soigneusement analysé avant d'être publié. Les chercheurs ont étudié d'autres modèles d'attaques, où l'exécution du code sur le matériel de la victime est plus facile. Un attaquant peut attaquer la victime à distance, souvent via le web, et monter des attaques sur le cache à travers le réseau [BB05, KGA+20]. Schwarz et collab. [SSL+19] ont montré qu'il est possible de monter une attaque Spectre via le web. Les environnements d'informatique en nuage sont également une cible idéale pour les attaques microarchitecturales. Avec le développement de l'infrastructure en tant que service, les utilisateurs peuvent acheter des machines virtuelles ou des conteneurs fonctionnant sur des services comme Amazon EC2 ou OVHCloud. Toutefois, l'informatique en nuage n'est qu'une couche d'abstraction pour des serveurs colossaux. Bien qu'isolées d'un point de vue purement logiciel, les machines virtuelles contrôlées par différents utilisateurs peuvent être co-résidentes de processeur, c'est-à-dire partager un ou plusieurs processeurs. Un attaquant peut donc acheter de l'espace dans le nuage et monter des attaques sur toutes les machines co-résidentes. Ce scénario a été exploité pour détecter la co-résidence [ZJOR11], attaquer l'ASLR [BRPG15] ou monter des canaux à large bande passante entre deux machines virtuelles isolées [MWS+17].

En 2015, Oren et collab. [OKSK15] ont présenté une attaque sur le cache s'exécutant entièrement dans le navigateur. Ce modèle de menace augmente considérablement la surface d'attaque des canaux auxiliaires microarchitecturaux. Comme le web moderne est construit autour de pages web dynamiques et interactives, l'architecture des sites web est séparée en deux composants majeurs. Le côté serveur gère tous les calculs concernant le serveur, la base de données, l'authentification et l'envoi des informations de la page web aux utilisateurs. Le côté client gère la page web telle qu'elle s'affiche dans le navigateur de l'utilisateur. Les

langages de script côté client, tels que JavaScript, sont responsables des composants interactifs des sites. Le code côté client est envoyé à l'utilisateur par le serveur et est exécuté dans le navigateur de l'utilisateur, et donc *est exécuté sur la machine de l'utilisateur*. Cela permet à un attaquant d'exécuter un script sur de nombreuses victimes différentes sans effort. Un utilisateur visitant un site web contrôlé par un attaquant téléchargera du code JavaScript et l'exécutera dans son navigateur sans s'en rendre compte, devenant ainsi potentiellement victime d'attaques microarchitecturales. Plus inquiétant encore, un attaquant peut acheter de l'espace publicitaire sur un site web légitime et exécuter le code JavaScript sur tous les utilisateurs qui voient la publicité.

Pour des raisons de sécurité, le code JavaScript s'exécute dans un environnement virtualisé sécurisé de type *bac à sable*. Ce bac à sable empêche l'accès aux adresses mémoire, aux instructions natives ou au système de fichiers. En outre, JavaScript est un langage abstrait de haut niveau, et il est difficile de quantifier son impact microarchitectural. Ces couches de sécurité et d'abstraction rendent difficile la mise en place de canaux auxiliaires microarchitecturaux à partir du bac à sable. Avec son attaque sur le cache en JavaScript, Oren et collab. [OKSK15] ont démontré la possibilité de provoquer et de mesurer des fuites de données microarchitecturales depuis le navigateur. Les caches ne sont pas le seul composant microarchitectural visé, et des chercheurs ont monté des attaques sur la DRAM [SMGM17] ou les unités à virgule flottante [AKM+15]. Bien qu'en général, les canaux auxiliaires JavaScript aient tendance à offrir moins de résolution aux attaquants, ils représentent une menace inquiétante pour la sécurité et la vie privée des utilisateurs. Les canaux auxiliaires microarchitecturaux ont été exploités dans le navigateur pour extraire des clés cryptographiques [GPTY18], surveiller les sites parcourus par un utilisateur [OKSK15, SKH+19], identifier les utilisateurs [LMD+22] ou communiquer en dehors du bac à sable [SMGM17].

Bien que les canaux auxiliaires basés sur le JavaScript aient été grandement étudiés, l'ensemble de leurs implications en matière de sécurité et de confidentialité n'est pas clair. En particulier, plusieurs zones grises subsistent :

Q1 Depuis les premières attaques sur le cache basées sur le JavaScript, les navigateurs ont évolué, notamment en réaction aux attaques par minutage. Quel est l'état actuel des attaques par minutage dans sur les navigateurs et leurs contre-mesures ? En particulier, les contre-mesures basées sur les minuteurs sont une approche populaire. Cependant, ces mesures ont souvent été adoptées dans l'urgence. Quel est leur impact sur la sécurité et la confidentialité des navigateurs ?

Q2 Quels autres composants sont vulnérables aux canaux auxiliaires depuis la sandbox JavaScript ?

Q3 Quelles informations un attaquant peut-iel extraire de ces canaux auxiliaires ?

Contributions

Cette thèse se concentre sur l'exploration partielle des réponses à ces questions. Les contributions de ce manuscrit sont divisées en deux catégories. La première direction de recherche consistait à redéfinir la portée des attaques par minutage en JavaScript. En particulier, nous avons étudié l'impact de contre-mesures répandues aux attaques par minutage : la suppression de l'accès aux minuteurs à haute résolution. La deuxième catégorie de contributions concerne un type spécifique de canal auxiliaire : la contention des ports du microprocesseur. Dans une

première contribution, nous montrons comment, pour la première fois, nous avons implémenté la contention de port dans le bac à sable du navigateur. Dans une deuxième contribution, nous étendons le champ d'application de la contention de port pour changer complètement son modèle de menace. Nous montrons également comment cette nouvelle version du canal auxiliaire peut être utilisée à des fins d'identifications des navigateurs.

Évaluation méthodique des minuteurs JavaScript (Q1) En réaction à la menace croissante que représentent les attaques par minutage basées sur JavaScript, les développeurs de navigateurs ont proposé diverses contre-mesures. Au milieu de tous ces changements, il peut être difficile de suivre les différentes évolutions des navigateurs. En particulier, l'impact des contre-mesures actuelles sur les attaques décrites dans la littérature n'est pas clair.

Nous présentons l'évolution des attaques par minutage dans les navigateurs, et fournissons des outils statistiques pour caractériser les minuteurs disponibles. Notre objectif est de présenter une vue claire de la surface d'attaque et de comprendre quelles sont les principales conditions préalables et classes d'attaques par minutage dans les navigateurs et quelles sont les principales contre-mesures. Nous fournissons une classification des attaques par minutage dans les navigateurs, en soulignant leurs conditions préalables communes. Nous proposons ensuite une taxonomie des contre-mesures en fonction des ressources qu'elles ciblent. Ces classifications permettent de déterminer dans quelle mesure les changements récents dans les contre-mesures ont un impact sur la sécurité des navigateurs.

Nous nous intéressons particulièrement à une contre-mesure très répandue : la suppression des minuteurs à haute résolution. L'idée est simple : si les attaquants ne peuvent pas mesurer les différences de temps qui créent les fuites des données, ils ne peuvent pas monter leur attaque par minutage. Nous avons créé des outils analytiques pour évaluer la menace que représente un minuteur. Nous avons développé un système automatique pour évaluer la sécurité des minuteurs dans différentes versions des navigateurs les plus populaires. En particulier, nous montrons que l'évolution de la protection contre les attaques par exécution transientes a rendu possible d'autres attaques, telles que les attaques par canal auxiliaire microarchitectural avec une bande passante plus importante que ce qui était possible il y a seulement quelques années.

Ce travail est le résultat d'une collaboration avec Clémentine Maurice (Univ Lille, CNRS, Inria) et Pierre Laperdrix (Univ Lille, CNRS, Inria). Il a été publié à EuroS&P 2021 [RML21].

Contention de port dans les navigateur web (Q2,Q3) La contention des ports du processeur est un canal auxiliaire microarchitectural introduit par Aldaya et collab. [ABuH+19] en 2019. Il utilise les ports du CPU, un composant du pipeline d'exécution, comme goulot d'étranglement pour créer des différences d'exécution et faire fuir des données confidentielles.

Dans cet article, nous présentons le premier canal auxiliaire de contention de port s'exécutant entièrement dans un navigateur web, malgré un environnement très restreint :

C1 Les minuteurs JavaScript ont une résolution inférieure à celle des minuteurs matérielles natives, ce qui augmente le bruit de mesure de l'attaquant.

C2 L'attaquant n'a aucun contrôle sur le coeur physique sélectionné par l'ordonnanceur pour exécuter le code d'attaque.

C3 Dans ce contexte, le code de l'attaquant est écrit dans un langage hautement abstrait qui est converti en code machine par un compilateur.

Alors que **C1** a été étudié par des travaux antérieurs [SMGM17, RML21], **C2** et **C3** nécessitent de nouvelles approches. Pour résoudre **C2**, nous proposons une heuristique entièrement basée sur JavaScript pour permettre la co-résidence de coeur entre l'attaquant et la victime. **C3** est probablement le plus grand défi : nous ne savons pas comment notre code de haut niveau sera traduit en code machine, donc son impact sur la microarchitecture et plus spécifiquement sur les ports du processeur n'est pas clair. Dans ce contexte, nous fournissons un système pour évaluer la contention de port causée par les instructions WebAssembly sur les processeurs Intel, permettant d'augmenter la portabilité des canaux auxiliaires de contention de port. Nous avons trouvé plus de 100 instructions créant de la contention sur 4 ports différents sur des microprocesseurs x86.

Notre attaque peut être utilisée pour construire un canal caché inter-navigateurs avec un débit de 200 bit/s, un ordre de grandeur au-dessus de l'état de l'art. Ce canal caché est inquiétant car il brise le modèle de sécurité d'isolation fondamental des navigateurs, permettant à deux onglets d'échanger des informations de cookies ou de communiquer avec un processus natif pour extraire des données privées. La contention de port web a une résolution spatiale de 1024 instructions natives dans une attaque par canal auxiliaire, ce qui est comparable aux meilleures attaques par cache dans le navigateur.

Nous concluons de notre travail que les attaques par contention de port sont non seulement rapides, mais sont aussi moins sensibles au bruit que les attaques par cache, et sont immunisés contre les contre-mesures implémentées dans les navigateurs ainsi que la plupart des contre-mesures par canal auxiliaire, qui ciblent le cache dans leur grande majorité.

Ce travail est le résultat d'une collaboration avec Clémentine Maurice (Univ Lille, CNRS, Inria), Marina Botvinnik (Ben-Gurion University of the Negev), et Yossi Oren (Ben-Gurion University of the Negev). Il a été publié à AsiaCCS 2022 [RMBO22].

Port Contention sans SMT (Q3) L'un des principaux prérequis de Port Contention est qu'il exploite un composant dépendant du coeur qui doit être partagé entre l'attaquant et la victime. De ce fait, elle s'appuie fortement sur SMT, une technologie permettant de partager un cœur physique entre plusieurs threads. Cependant, certains microprocesseurs n'utilisent pas le SMT, ou certains systèmes d'exploitations le désactivent pour des raisons de sécurité, à l'instar de ChromeOS [Goob] ou RedHat [Lar].

Dans cet article, nous présentons la *contention séquentielle de port*, qui ne nécessite pas de SMT. Au lieu d'exploiter le parallélisme *au niveau du fil d'exécution*, nous exploitons le parallélisme *au niveau de l'instruction*. Il exploite l'ordonnancement sous-optimal des ports d'exécution pour la parallélisation au niveau des instructions. Par conséquent, des séquences d'instructions spécifiquement conçues sur un seul thread souffrent d'une latence accrue. Nous démontrons une mise en œuvre de la contention de port séquentielle native sur les processeurs x86.

Nous montrons que la contention de port séquentielle peut être exploitée à partir des navigateurs web en WebAssembly, y compris sur les navigateurs orientés vers la confidentialité tels que Tor Browser ou Brave. Nous présentons un système automatisé pour rechercher les séquences d'instructions menant à la contention de port séquentielle pour des générations de microprocesseurs spécifiques, que nous avons évalué sur 50 microprocesseurs différents, y compris x86 et AMD. Un attaquant peut utiliser ces séquences à partir du navigateur pour déterminer la génération de processeur dans les 12 s avec une précision de 92%. Cette empreinte digitale est très stable dans le temps et résistante au bruit du système, ce qui la

rend très précieuse pour compléter les empreintes digitales basées sur des attributs logiciels plus volatiles. En outre, nous montrons que les mesures de protection sont soit coûteuses, soit uniquement probabilistes.

Ce travail est le résultat d'une collaboration avec Clémentine Maurice (Univ Lille, CNRS, Inria) et Michael Schwarz (CISPA Helmholtz Center for Information Security). Il a été publié à ESORICS 2022 [RMS22].

Contents

1. Introduction **1**

List of Productions **9**

2. Background **11**
2.1. CPU Overview 12
2.2. Web Browsers 20
2.3. High-Resolution Timers 24
2.4. Microarchitectural Attacks 26
2.5. Side-Channel Attacks on Software and Browser Resources 42
2.6. Countermeasures to Side Channels 44
2.7. Browser Fingerprinting 51

3. High Resolution Timers in the Browser **53**
3.1. Timing attacks in browsers 54
3.2. Countermeasures in browsers 57
3.3. Evaluation tools 58
3.4. Results 62
3.5. Discussion 69
3.6. Conclusion 71
3.7. Evolution of Timer Security Since The Publication of the Results 71

4. Port Contention in the Browser **73**
4.1. Web-Assembly-Based Port Contention 74
4.2. PC-detector 77
4.3. Side-channel Attack on Artificial Applications 79
4.4. Covert Channel 81
4.5. Discussion 87
4.6. Conclusion 88

5. Port Contention Without SMT and its Privacy Implications **89**
5.1. Threat Model 90
5.2. Port Contention Without SMT 90
5.3. Fingerprinting CPU Generations 94
5.4. Discussion 98

5.5. Conclusion 100

6. Conclusion and Perspectives **101**

A. Appendices **105**
A.1. Custom RDTSC implementation 105
A.2. Port Contention on Other WebAssembly Instructions 107
A.3. Training set 108

Bibliography **109**

1 Introduction

Modern processors are remarkably optimized pieces of technology. As processing speed is probably the major selling point of CPUs, vendors have focused on increasing processors' performances in their development, each year bringing new generations with better features than the last.

As the progress in semiconductors' size and performances slowed industry-wide over the last decades, simply adding more transistors to a chip became insufficient to follow a steady performance rise. In reaction, CPU vendors focused on microarchitectural optimizations to further improve performance from generation to generation. These optimizations aim at increasing processing speed by proposing more parallelism and reducing stall times of the hardware. Examples of microarchitectural optimizations include multi-core processors, caches, hyperthreading, or speculative execution. Each generation sees changes to the microarchitecture, bringing more and more buffers or units in a complex pipeline. By design, these optimizations aim to bring more performance by speeding the execution. The security aspect of the microarchitecture tends to be secondary. In particular, side effects of these optimizations on computation may leak information about secret values used by the system in what are called *side channels*.

In 1996, Paul Kocher [Koc96] discovered that the dynamic speedup brought by caches introduces runtime differences in constant-time implementations of cryptographic algorithms. A malicious user can use these subtle timing differences to extract private information such as decryption keys on Diffie-Hellman. This discovery of the first timing attack paved the way for the broader research field on side channels.

Physical side channels have been heavily investigated. They exploit the physical side effects of the processor during the computation to detect sensible computation. The first physical side channel leveraged the power consumption of the chip [KJJ99] to attack cryptographic implementations, but many sources of physical leakage have been studied, including acoustic emissions [ST04, GST14], electromagnetic emissions [RR01], optical emissions [SNK+12] or temperature [HS13]. However, except for notable exceptions [LKO+21], these attacks require physical access to the target device to measure the side effects of computation.

In constrast, *microarchitectural side channels* are purely software-based. They exploit the data-based optimizations of modern processors to infer private information. Since Kocher's first timing attack [Koc96], caches have probably been the most studied source of microarchitectural side channels, but research has proven that other components can also be exploited to mount side-channel attacks. Over the last decade, researchers have picked many microarchitectural components, reversed them, and used their behavior to extract secrets. Side channels have been leveraged, among others, out of branch predictors [AKS07], DRAM [PGM+16], or address translation [GRBG18]. Microarchitectural side channels were initially mounted to leak data from constant-time cryptographic implementations, but their

scope has vastly broadened, including attacks against kernel-space ASLR [EPA16], cross-VM communication [MWS+17], keystroke detection [SLG+18b] and user monitoring [SKH+19]. Even more worrisome, microarchitectural side channels are an essential building block of a new class of microarchitectural attacks: transient execution attacks. Introduced in 2018 with Spectre and Meltdown, these attacks use the side effects of transient instructions, i.e., instruction executed but never committed to the architecture, to leak data through software and hardware security barriers.

Microarchitectural side channels exploit the sharing of hardware components between the attacker and the victim to leak information. Consequently, they have one common prerequisite: *the attacker must run code on the victim's machine.* This essential precondition brings severe restrictions to the attack surface of microarchitectural side channels, as system users grow increasingly wary of running unknown code on their personal machines. Awareness campaigns about the risk of unknown code have reached the ears of personal computer users, who tend to only download software from trusted sources. The risk of downloading attacker code is even reduced on mobile platforms, where applications are downloaded from an editor-controlled marketplace, each piece of code being carefully analyzed before release. Researchers have studied other threat models, where running code on the victim's hardware is elementary. An attacker can attack the victim remotely, often through the web, and mount network cache attacks [BB05, KGA+20]. Schwarz et al. [SSL+19] showed that it is possible to mount a Spectre attack over the web. Cloud environments are also an ideal target for microarchitectural attacks. With the development of Infrastructure as a Service (IaaS), users can buy Virtual Machines (VM) or containers running in the cloud on services like Amazon EC2 or OVHCloud. However, the cloud is just an abstraction layer for colossal servers. Although isolated from a purely software point of view, VMs controlled by different users can achieve processor co-residency, i.e., share one or more processors. This means an attacker can buy space in the cloud and mount attacks on all co-resident machines. This scenario has been exploited to detect co-residency [ZJOR11], reversing ASLR [BRPG15] or mounting high-bandwidth channels between two isolated virtual machines [MWS+17].

In 2015, Oren et al. [OKSK15] presented a cache attack running entirely in the browser. This threat model considerably enhances the attack surface of microarchitectural side channels. As the modern web is built around dynamic and interactive web pages, the architecture of websites is separated into two major components. The server side, or back-end, handles all computations regarding the server, the database, authentication, and sending the webpage information to the users. The client side, or front-end, handles the webpage as displayed in the user's browser. Client-side scripting languages such as JavaScript are responsible for the interactive components of sites. Client-side code is sent to the user by the server and is executed in the user's browser, and therefore *is executed on the user's machine.* This allows an attacker to run arbitrary code on many different victims effortlessly. A user visiting an attacker-controlled website will download JavaScript code and run it in his browser without noticing, potentially falling victim to microarchitectural attacks. Even more worrisome, an attacker can buy advertisement space on a legitimate website and run JavaScript code on all users seeing the ad.

For security reasons, JavaScript code runs in a secured virtualized environment called a sandbox. This sandbox prevents access to memory addresses, native instructions, or file system accesses. Furthermore, JavaScript is a high-level abstracted language, and it is hard to quantify its microarchitectural impact. These layers of security and abstraction make it hard to mount microarchitectural side channels from the sandbox. With their JavaScript cache attack, Oren

et al. [OKSK15] demonstrated the possibility of causing and measuring microarchitectural data leakage from the browser. Caches are not the only microarchitectural component targeted, and researchers have mounted attacks on DRAM [SMGM17] or floating-point units [AKM+15]. Although in general, JavaScript side channels tend to offer less resolution to the attackers, they represent a worrying threat to users' security and privacy. Microarchitectural side channels have been leveraged in the browser to extract cryptographic keys [GPTY18], monitor sites browsed by a user [OKSK15, SKH+19], fingerprint users [LMD+22] or communicate outside of the sandbox [SMGM17].

Although JavaScript-based side channels have been studied by academia, the entire landscape of their security and privacy implications is unclear. In particular, several gray areas remain:

Q1 Since the first JavaScript-based cache attacks, browsers have evolved, including in reaction to timing attacks. What is the current landscape of browser-based attacks and countermeasures?

Q2 Are other components vulnerable to side channels from the JavaScript sandbox?

Q3 What can an attacker extract from these side channels?

Contributions

This thesis is focused on partially exploring the answers to these questions. The following contributions of this manuscript are split into two categories. The first research direction was to reframe the scope of JavaScript-based timing attacks. In particular, we studied the impact of widespread countermeasures to timing attacks: removing access to high-resolution timers. The second class of contributions concerns a specific type of side-channel: CPU port contention. In a first contribution, we show how, for the first time, we implemented port contention in the browser sandbox. In a second contribution, we extend the scope of port contention to change its threat model completely. We also show how this new version of the side channel can be used in browser fingerprinting.

Systematic evaluation of JavaScript timers (Q1) In reaction to the developing threat posed by JavaScript-based timing attacks, browser vendors have proposed various countermeasures. However, as the attack multiplied in the last years, so did the countermeasures, in a cat-and-mouse game fashion. Amid all these changes, it can be hard to keep track of all the different evolutions browsers underwent. Notably, it is unclear how current countermeasures impact the attacks described in the literature.

We present the evolution of timing attacks in browsers, and provide statistical tools to characterize available timers. Our goal is to present a clear view of the attack surface and understand what are the main prerequisites and classes of browser-based timing attacks and what are the main countermeasures. We provide a classification of browser-based timing attacks, highlighting their common prerequisites. We then propose a taxonomy of countermeasures based on the resources they target. These classifications focus on determining to what extent the recent changes in countermeasures impact browser security.

We take a particular interest in a widespread countermeasure: removing high-resolution timers. The idea is simple: if attackers cannot measure data-leaking timing differences, they

cannot mount their timing attack. We created analytical tools to evaluate the threat posed by a timer. We developed an automatic framework to evaluate timer security in different versions of the most popular browsers. In particular, we show that the shift in protecting against transient execution attacks has re-enabled other attacks such as microarchitectural side-channel attacks with a higher bandwidth than what was possible just a few years ago.

This work is the outcome of a collaboration with Clémentine Maurice (Univ Lille, CNRS, Inria) and Pierre Laperdrix (Univ Lille, CNRS, Inria). It has been published in the proceedings of EuroS&P 2021 [RML21].

Port Contention in the Web Browser (Q2,Q3) CPU Port contention is a microarchitectural side-channel introduced by Aldaya et al. [ABuH+19] in 2019. It uses CPU ports, a component of the execution pipeline, as a bottleneck to create runtime differences and leak confidential data.

In this contribution, we present the first port contention side channel running entirely in a web browser, despite a highly challenging environment:

C1 Web-based timers have a lower resolution than native hardware-based timers, increasing the attacker's measurement noise.

C2 The attacker has no control over the physical core selected by the browser to execute the attack code.

C3 In this setting, the attacker's code is written in a highly-abstracted language which is translated into machine code by a just-in-time compiler.

Whereas **C1** has been studied by previous work [SMGM17, RML21], **C2** and **C3** require new approaches. To solve **C2**, we propose a fully JavaScript-based heuristic to allow core co-residency between the attacker and the victim. **C3** is probably the biggest challenge: we do not know how our high-level code will be translated into machine code, thus its impact on the microarchitecture and more specifically on the CPU ports is unclear. To that extent, we provide a framework to evaluate the port contention caused by WebAssembly instructions on Intel processors, allowing to increase the portability of port contention side channels. We found over 100 instructions creating contention on 4 different ports on x86 CPUs.

Our attack can be used to build a cross-browser covert channel with a bit rate of 200 bit/s, one order of magnitude above state of the art. This covert channel is worrisome as it breaks the fundamental isolation security model of browsers, allowing two tabs to exchange cookie information or communicate with a native process to extract private data. Web port contention has a spatial resolution of 1024 native instructions in a side-channel attack, performing on-par with the best cache attacks in the browser.

We conclude from our work that port contention attacks are not only fast, but are also less susceptible to noise than cache attacks, and are immune to countermeasures implemented in browsers as well as most side-channel countermeasures, which target the cache in their vast majority.

This work is the outcome of a collaboration with Clémentine Maurice (Univ Lille, CNRS, Inria), Marina Botvinnik (Ben-Gurion University of the Negev), and Yossi Oren (Ben-Gurion University of the Negev). It has been published in the proceedings of AsiaCCS 2022 [RMBO22].

Port Contentiont Without SMT (Q3) A major prerequisite of Port Contention is that it exploits an on-core component that must be shared between the attacker and the victim. To that extent, it heavily relies on simultaneous multi-threading (SMT), a technology allowing to share a physical core between several threads. However, certain CPUs do not implement SMT, or operating systems disable it for security purposes, such as ChromeOS [Goob] or RedHat [Lar].

In this contribution, we present *sequential port contention*, which does not require SMT. Instead of exploiting *thread-level* parallelism, we exploit *instruction-level* parallelism. It leverages sub-optimal scheduling to execution ports for instruction-level parallelization. As a result, specifically-crafted instruction sequences on a single thread suffer from an increased latency. We demonstrate an implementation of native sequential port contention on x86 processors.

We show that sequential port contention can be exploited from web browsers in WebAssembly, including on privacy-oriented browsers such as Tor Browser or Brave. We present an automated framework to search for instruction sequences leading to sequential port contention for specific CPU generations, which we evaluated on 50 different CPUs, including x86 and AMD. An attacker can use these sequences from the browser to determine the CPU generation within 12 s with a 92% accuracy. This fingerprint is highly stable in time and resistant to system noise, making it highly valuable to complement more volatile software-attribute fingerprinting. Furthermore, we show that mitigations are either expensive or only probabilistic.

This work is the outcome of a collaboration with Clémentine Maurice (Univ Lille, CNRS, Inria) and Michael Schwarz (CISPA Helmholtz Center for Information Security). It has been published in the proceedings of ESORICS 2022 [RMS22].

Terminology

The terminology of classes of attacks explored in this manuscript can be confusing, as classes intertwine and denominations can be orthogonal. This manuscript is generally focusing on three major classes of attacks:

Timing Attacks: They exploit timing differences caused by the execution to infer private data. They can be microarchitectural timing attacks, e.g., cache attacks, but also purely based on timing differences created by software execution.

Side Channels: They exploit the side effects of execution to infer information about the victim. Timing attacks typically are side-channel attacks, but side channels may exploit other side effects than time differences. They can be physical side channels, e.g., electromagnetic side channels, microarchitectural, e.g., cache attacks, or purely software-based.

Microarchitectural attacks: Microarchitectural attacks exploit the microarchitectural insecure implementation to break security rings. They comprehend microarchitectural side channels, microarchitectural fault attacks, and transient execution attacks.

Figure 1.1 illustrates the relations between these classes of attacks. This manuscript takes a particular interest in microarchitectural side channels with port contention [RMBO22, RMS22] and timing attacks in the browser [RML21].

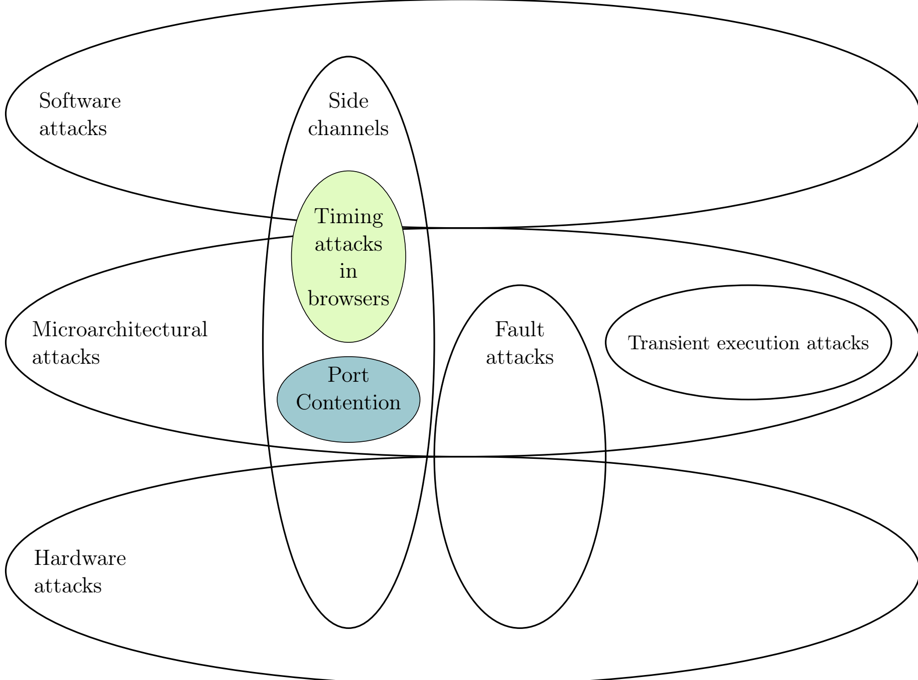


Figure 1.1. – Visualisation of various attack classes mentioned in this manuscript, and scope of our publications

Artifacts

Reproducible research is an essential value of this thesis. To that extent, we released all source code and evaluation data for our works. In particular, JavaScript exploits are highly portable by nature, and all client-side attacks can be run in the latest versions of browsers as of 2022. We also provide the various frameworks proposed in that work, hoping it can bring a more systematic approach to web-based side channel research. The artifacts are separated into three repositories:

- Framework, modified browsers, and evaluation data on our survey of timer security [RML21] are available on `https://github.com/thomasrokicki/in-search-of-lost-time`.
- PoCs and framework of our web-based port contention implementation [RMBO22] are available on `https://github.com/MIAOUS-group/web-port-contention`.
- PoCs, framework, and classification models for sequential port contention [RMS22] are available on `https://github.com/MIAOUS-group/port-contention-without-smt`.

Thesis Outline

The rest of this manuscript is divided into 5 chapters:

- Chapter 2 introduces some fundamental notions to understand the contributions of this manuscript. In particular, Section 2.1 introduces hardware notions about modern

processors, especially Intel CPUs. Section 2.2 describes the architecture of web browsers. Section 2.3 provides a formalization of timers, and examples of timers in native and web environments. Section 2.4 introduces state of the art on microarchitectural attacks while Section 2.5 presents a short overview of other timing attacks with a strong focus on the browser. Section 2.6 presents the different classes of countermeasures proposed by the industry and academia. Finally, Section 2.7 briefly introduces browser fingerprint.

- Chapter 3 details our work on a systematic approach to timing attacks and timers in web browsers. It provides a taxonomy of browser timing attacks and countermeasures, and statistical tools to evaluate timer security in recent versions of browsers.
- Chapter 4 describes our implementation of port contention in the browser's sandbox. It also provides several applications of this side channel: a high-bandwidth covert channel and a side-channel artificial example.
- Chapter 5 introduces sequential port contention, a new development of port contention with fewer prerequisites. We show how sequential port contention can be used in web environments and its implication on the user's privacy via an application on browser fingerprinting.
- Chapter 6 concludes this thesis and opens perspectives to future work.

1 List of Productions

[RML21] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. Sok: In search of lost time: A review of javascript timers in browsers. In *EuroS&P*, pages 472–486. IEEE, 2021. Artifacts: https://github.com/thomasrokicki/in-search-of-lost-time

[RMBO22] Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. Port contention goes portable: Port contention side channels in web browsers. In *AsiaCCS*, pages 1182–1194. ACM, 2022. Artifacts: https://github.com/MIAOUS-group/web-port-contention

[RMS22] Thomas Rokicki, Clémentine Maurice, and Michael Schwarz. CPU port contention without SMT. In *ESORICS*, 2022. Artifacts: https://github.com/MIAOUS-group/port-contention-without-smt

Background 2

In this chapter, we provide the necessary background to appreciate and discuss the contributions of this manuscript on microarchitectural attacks, especially in the web browser.

It is decomposed in 7 sections:

- Section 2.1 provides an overview of the execution pipeline and memory subsystem of modern Intel processors.
- Section 2.2 presents the architecture of a web browser.
- Section 2.3 introduces an important notion for this manuscript: high-resolution timers. In particular, it explains several techniques to build high-resolution timers in web browsers.
- Section 2.4 explores the state of the art on microarchitectural attacks. In particular, we present microarchitectural side channels, fault attacks and transient attacks. For each class, we present native attacks and attacks running in the browser.
- Section 2.5 presents other timing attacks exploiting software resources, with a strong emphasis on browser-based attacks.
- Section 2.6 explores state of the art countermeasures to side channels.
- Finally, Section 2.7 briefly introduces browser fingerprinting.

2.1. CPU Overview

This section presents a high-level overview of a modern Intel processor. CPUs are typically built in a multi-core architecture. These *physical cores* are highly independent and can process data in parallel, effectively multiplying performances. Each core contains its own execution pipeline, able to fetch instructions and execute them, as well as an on-core memory subsystem. Cores communicate with other cores through a ring interconnect. Some components, e.g., the RAM are shared by all physical cores.

2.1.1. Execution Pipeline

Pipelining is an essential model of modern processor architecture. A simplified RISC architectural pipeline can be decomposed into 5 stages:

Instruction Fetch: fetches, i.e., loads, an instruction in the pipeline.

Instruction Decode: translates this instruction and address registers.

Execute: sends the instruction to execution units and executes it.

Memory: read from memory to a register, or write a register to memory.

Writeback: store the results in a register, committing changes to the architecture.

The main idea of the pipeline is that to retrieve maximal performances, all stages of the pipeline must function at all times. At the first clock cycle, the pipeline fetches instruction i_0. Then, at the next cycle, i_0 is decoded while the pipeline fetches instruction i_1. On standard execution, each pipeline stage handles an instruction per cycle. If a stage does not handle an instruction in a cycle, this cycle is effectively lost in terms of performance. This stall propagates to the rest of the pipeline, and is designated as a bubble. Bubbles can be introduced by unavailable resource or by waiting for dependencies. Modern processor implement several mechanism to reduce this stalling times, such as out of order execution or branch prediction.

The microarchitectural pipeline is the implementation of this architectural pipeline. Figure 2.1 illustrates Intel's Skylake CPUs execution pipeline. It is separated into a *front end*, handling fetch and decode stages, as well as a *back end*, handling the execute, memory, and writeback stages. Each stage is highly parallelized, meaning several instructions are fetched, decoded, and executed every cycle.

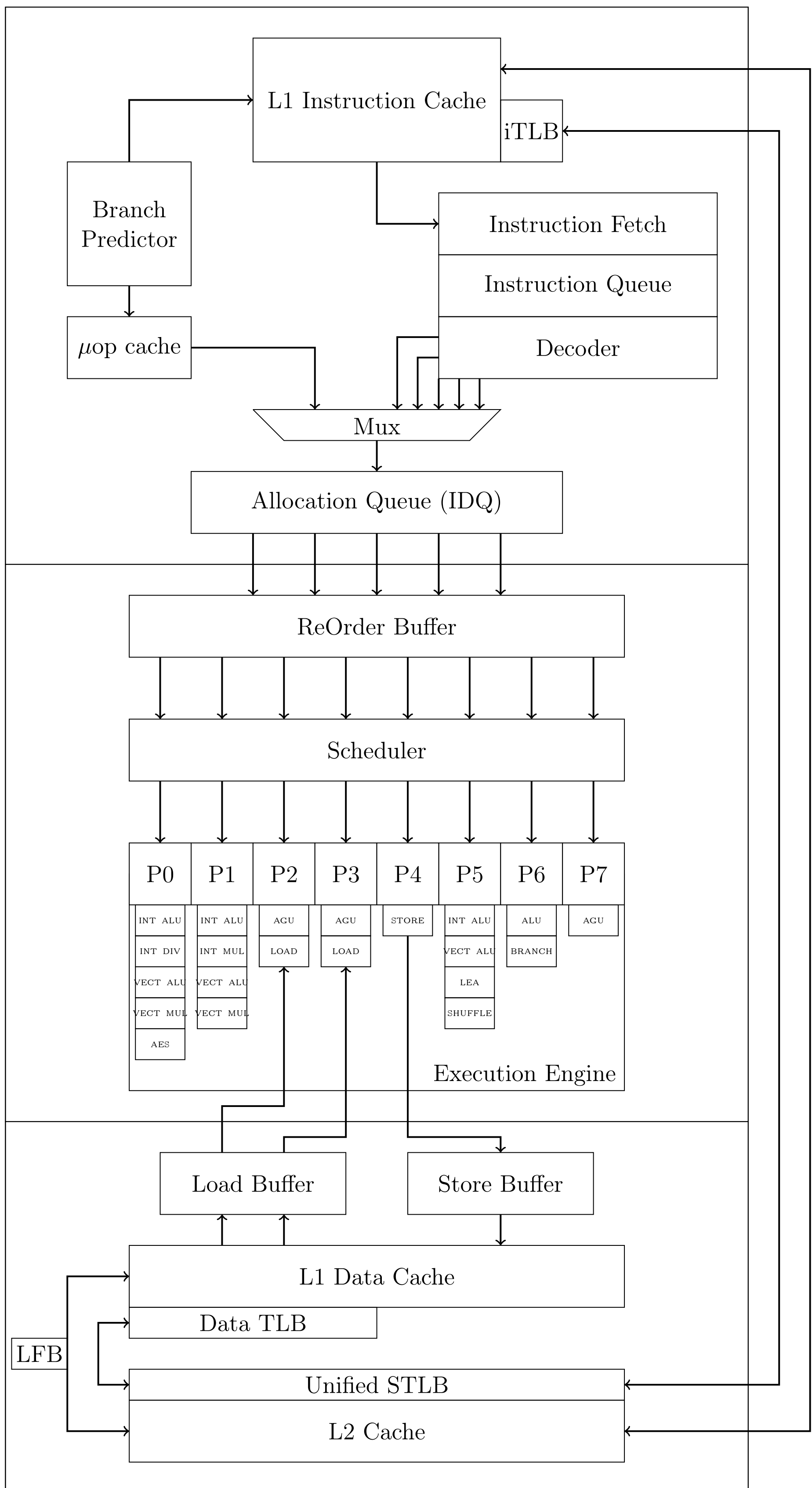


Figure 2.1. – Simplification of the execution pipeline and memory subsystem for a single core in the Skylake microarchitecture.

2.1.1.1. Front End

The front end is responsible for fetching and decoding instructions and forwarding them to the execution engine or back end. The Instruction Fetch Unit fetches data from the L1 instruction cache, reading packets of 16 bytes. These packets are then split into x86 instructions by the predecoder. These macro-operations are then forwarded to the FIFO Instruction Queue (IQ). The IQ implements macro-op fusion. Certain combinations of x86 instructions, e.g., a compare test and a subsequent jump, are merged into a single x86 instruction, saving bandwidth in the rest of the pipeline. At this point, instructions vary greatly in size and can have inconsistent encoding. The decoder decomposes instructions in atomic, fixed-length operations, called *micro-operations* or *μops*. This decomposition is deterministic, i.e., a given instruction is decomposed in the same μops regardless of the context. Abel and Reineke [AR19] have characterized the decomposition of x86 instructions[1]. They propose a benchmarking tool empirically measuring the number of μops decomposed from the instruction, the instruction's latency, i.e., the number of cycles required to execute all the instruction's μops [Intc] or the instruction's throughput, i.e., the average number of cycles per instruction when the instruction is repeatedly called [F+11]. The μops are then sent to the Instruction Decode Queue (IDQ).

Data or control dependencies can introduce pipeline stalls, reducing the processor's overall performance. For instance, when executing a conditional branch, e.g., if/else instructions, the pipeline has to wait for the conditional instruction to be executed and committed to the architecture before knowing which branch to take. To reduce control-dependencies-based delay, modern CPUs implemented branch prediction. Intel processors handle it with the Branch Prediction Unit (BPU). The BPU predicts, among others, the branch taken in a conditional branch. It implements a Pattern History Table (PHT) recording if a specific branch was taken or not in previous executions. The PHT is indexed according to virtual address bits and a Branch History Buffer (BHB), which stores branch decisions for all branches on the core. The advantage of the PHT is to remember which branch was taken on multiple sequential executions, allowing the identification of this pattern to increase the accuracy of predictions. For instance, if a branch is taken once every two executions, simply choosing the same branch as the last execution is insufficient. However, remembering two or more executions in the PHT allows to identify that pattern. The processor also predicts branch targets of direct or indirect branches before they are actually computed. The Branch Target Buffer (BTB) stores branch targets of previous branch executions. The BTB indexation scheme is not documented, but reverse-engineering [Hor] has shown that it uses a mixture of virtual-address bits of the branch instruction's address as well as previous, core-wide branches stored in the BHB. Return targets have a specific prediction component called the Return Stack Buffer (RSB). Similarly to the BTB, it dynamically stores the latest targets of RET instructions for a function and uses them for prediction. Mispredictions are only detected when the actual dependency is resolved. In that case, the pipeline is flushed, and the architecture is reset to its previous state, then executes the accurate branch.

Due to the inconsistent sizes and forms of x86 instructions, the fetching and decoding steps can be costly in performance and power. To optimize the front end, Intel introduced the Decode Stream Buffer (DSB) or μop cache. The μop cache dynamically stores the decomposition of the most frequent instructions in μops. It allows bypassing the costly decoding phase, and delivering μops to the IDQ faster. On Skylake architecture, the μop

[1] All the results can be found at uops.info

cache has an 80% hit rate [Intc], significantly boosting performances. Similarly to the standard decode queue, the μops are then pushed to the IDQ.

The IDQ implements an optimization known as Loop Stream Detector (LSD). It detects tight loops with many iterations and automatically fits all μops directly in the IDQ without fetching and decoding them. Such small-body-size[2] and high-iterations loops are often found in modern software, e.g., searches or string moves. It allows the prior stages of the front-end to serve on other threads, thus improving overall performances. The IDQ delivers its μops orderly to the back end.

2.1.1.2. Back End

At this point, μops from all threads are orderly stored in the IDQ and enter the back end. The goal of the back end is to execute the μops in the most optimized fashion possible. To take advantage of available resources, the backend promotes parallelism through *Out of Order exeuction* (or OoO). μops are not executed in their emission order, but rather as soon as all necessary resources are free.

μops are stored in the Re-Order Buffer (or ROB) until they are retired. The μops are sent to the scheduler to be executed. It queues the μops and forwards them to the scheduler. At the same level, the Branch Order Buffer (BOB) keeps track of the previous architectural states before speculation allowing for a fast roll-back in case of mispeculation. The ROB features many optimizations. Move Eliminations eliminates the μops of register-to-register moves, preventing latency in the back end. Zeroing Idioms, e.g., computing XOR over the same registers, are also eliminated from the ROB. Similarly, One Idioms, i.e., μops setting all bits of a register to 1 are removed from the ROB, effectively causing zero latency.

The instructions are then forwarded to the scheduler. The scheduler stores μops until all necessary resources are available e.g., the execution unit is ready, or all operands have been loaded. The scheduler then forwards μops to the execution units.

The execution engine is composed of several execution units, each specialized in a specific type of operations, e.g., algebraic operations for the ALUs. The number and types of execution units vary between microarchitectures. Execution units are grouped by *execution ports*. Each port leads to several specialized execution units; therefore, ports are also specialized. Figure 2.2 illustrates the 8-port structure for a Skylake processor. If possible, one μop is forwarded to each available port per cycle. As the decomposition of an x86 instruction in μops is deterministic, and each μop can only be executed by a specific type of execution unit, we can determine the *port usage* of an instruction, i.e., the ports used by this instruction. For instance, on Skylake processors, the IMUL R32 M32 instruction has a port usage of 1*p1+1*p23, i.e., emits a μop on P1 and a μop either on P2 or P3, as both ports can handle memory loads.

After their execution, μops are retired in an orderly fashion, and changes are committed to the architecture. All the pipeline resources are freed during retirement, including ROB and BOB entries or execution units. Skylake processors can commit up to 6 μops per cycle.

2.1.2. Hyperthreading

Modern CPUs are often separated into several physical cores. This boosts performance and grants more parallelism during computation, as each core has its own execution pipeline and low-level caches. However, most standard computation does not exploit all hardware

[2] typically 64 μops for Skylake

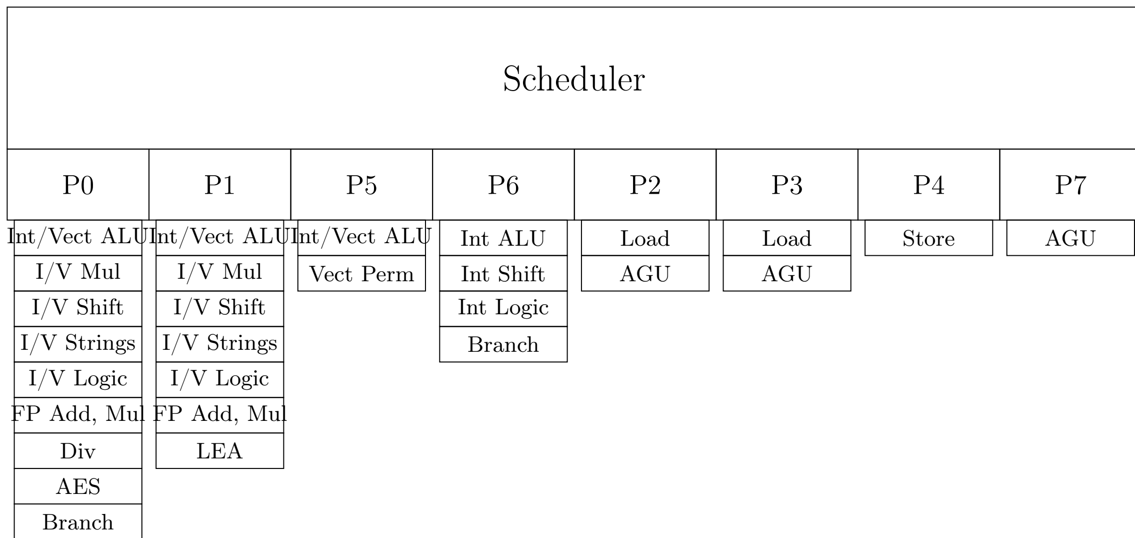


Figure 2.2. – Illustration of the execution unit distribution on ports for a Skylake microprocessor.

resources of a physical core at the same time. To exploit the hardware resources in the most optimized way possible, processor vendors have introduced Simultaneous MultiThreading (SMT). SMT allows multiple independent threads to use the physical resources of a physical core at the same time. The front end fetches instructions from all threads and execute them independently of their origin. All threads on the physical core share their execution pipeline, L1 and L2 caches, and bus. In addition, they have independent interruption systems and data and instruction registers. Sharing physical components between threads can be achieved in two settings:

Static Sharing: The component is definitively split, and a part of a component is only used by a single thread

Dynamic Sharing: The whole component is accessible by all threads competitively

This separation of physical cores is abstracted at the OS and software level. The OS considers each of these threads as a **logical core**, independent from the other logical cores. Hyperthreading is the name of Intel's implementation of SMT. It splits each physical core into two logical cores. On Skylake processors, the BPU or execution units are dynamically shared, whereas the IDQ, ROB, or μop cache are statically shared between threads.

2.1.3. Memory

During the execution, the CPU requires access to many values stored in the memory. On a single core, the system can handle 2 (P23) memory loads and 1 (P4) memory write per cycle for a Skylake processor. Access time to the memory is crucial as it can be one of the execution bottlenecks. The memory subsystem is highly optimized to reduce latency. This section presents an overview of the different components of the memory subsystem and shared memory focusing on caches as they are an essential component of microarchitectural attacks.

2.1.3.1. Virtual Memory

At the lowest level, RAM is indexed with physical addresses. However, processors introduced *virtual memory* to bring more isolation between different processes. Processes do not directly use the physical addresses but access them with a level of abstraction through *virtual addresses*. Modern virtual memory is implemented through pages, i.e., fixed-size memory blocks. The physical memory, respectively virtual memory, is split into physical pages, respectively virtual pages. Intel's processors typically use 4 kB pages, but larger pages can be requested by processes. Both physical and virtual pages are indexed through *page numbers*. Virtual and physical pages are aligned, i.e., the start of a virtual page is always the size of a physical page. This also means that, for 4 kB pages, the 12 least significant bits of a virtual address are equal to those of the physical address. The translation from virtual to physical address is handled by the Memory Management Unit (MMU). The virtual-physical mapping is decomposed in Page Tables (PT), and follows a 4 or 5-level hierarchical structure. When a processor wants to access a virtual address, the processor looks for the appropriate Page Table Entry (PTE) in Page Tables. This process is called a page walk. As almost any memory operation requires a translation, the translation latency is a significant bottleneck in modern systems. Intel implemented on-core Translation Lookaside Buffers (TLB) to lower this latency, dynamically caching the most frequent translations. A translation served from the TLB is significantly faster than from the standard PT structure. The TLB is often separated into an instruction TLB (iTLB) and a data TLB (dTLB).

Virtual pages are also the base unit for *Address-Space Layout Randomization (ASLR)*, a standard security measure to prevent code-injection attacks. When a process starts or requires new stack allocation, ASLR randomizes the address blocks. This implies that other processes cannot know which virtual memory pages are used by a specific process, thus preventing an attacker from modifying them. ASLR is also implemented on *kernel-space addresses (KASLR)*.

Sharing memory between processes is a critical optimization of the memory space. At the system level, the OS implements *memory deduplication*. It is implemented on a content-based approach: if several memory pages are identical, the OS frees all pages except one. This page is set as copy-on-write, i.e., shared by all processes generating this page before the duplication. When a process tries to write on the page, a page fault is thrown. The page is then duplicated, keeping the original page and the modified copy for the writing process.

2.1.3.2. Caches

Caches are small memories that sit close to the CPU core. They are relatively small, typically in the order of 1 to 10 MB. As they sit close to the execution pipeline, the access latency is significantly smaller than an access to the DRAM. The goal of caches is to dynamically store the most used values to serve most calls from the cache and not the DRAM, thus significantly improving performances.

Modern Intel CPUs often have three levels of caches of different sizes. The L1 cache is the smallest and fastest, while the L3, or last-level Cache, is the biggest and slowest. Both L1 and L2 are private to each physical core, whereas the last-level cache is shared by all cores. The last-level cache is *inclusive* to L1 and L2, meaning that all values in the L1 or L2 are also stored in the last-level cache.

Modern caches follow a set-associative organization illustrated in Figure 2.3. They are

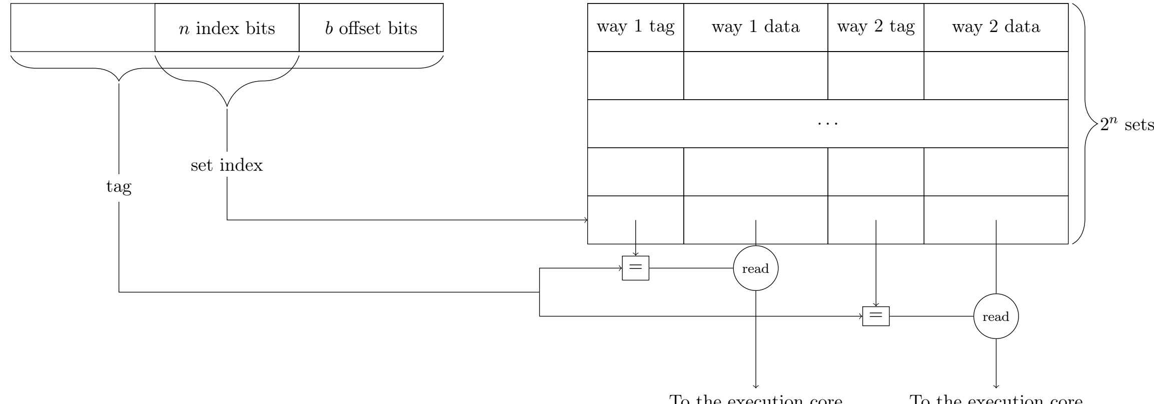


Figure 2.3. – Illustration of a 2-way set associative cache with 2^n cache sets

composed of 2^n cache sets, each composed of m ways. Such a cache is called an m-way associative cache. Each way consists of a tag and 2^b bytes of data. Most modern Intel caches use $b = 6$ and have 64-bytes lines. The tag is computed from the memory address stored in the cache line. The 6 least significant bits of the address are used as an *offset* in the data section of the line. The n middle bits of the addresses are used as a *set index*, indicating in which cache set the data must be stored. Two different addresses with the same set index are stored in the same cache set and are called *congruent* addresses. When the processor fetches an address from the cache, it first computes the set index to find the associated cache set. Then, it checks the tags from all the m lines present in the set and compare them with the tag computed from the wanted address. If one line has the sought tag, the data is served from the cache, causing a cache hit. If no line in the set has the wanted tag, the data is served from the DRAM, causing a cache miss.

The computation for the set index and the tag can be executed on the virtual address or the physical address. In a Virtually-Indexed Virtually-Tagged (VIVT) cache, both the tag and the set index are computed from the virtual address. This is interesting regarding latency, as the processor does not need to translate the virtual address into a physical one. However, this comes at a space cost, as shared memory is not shared in virtual addresses, meaning that some data could occupy several different cache lines. The opposite solution is to use a Physically-Indexed Physically-Tagged (PIPT) cache, where both the tag and the set are computed from the physical address. This grants a physically-unique tag, resulting in optimized use of cache space for shared memory but comes at a latency cost due to address translation. This is the most commonly used addressing mode, particularly in L2 and last-level caches for modern Intel processors. Finally, some vendors proposed a hybrid approach: Virtually-Indexed Physically-Tagged (VIPT) caches. This allows the processor to immediately compute the cache set index to remove the latency. The tag is then computed from the physical address while the set is looked up, thus reducing the latency. This mode is particularly interesting when the index does not use bits from a different page. This means, for a 4 kB page structure, the page offset is 12 bits long. As the line offset is computed of the 6 least significant bits, it leaves 6 bits to calculate the tag and stay on the same page offset, resulting in an optimal cache size of 2^8 cache sets. The L1 of most Intel processors follows this optimized structure and is composed of 2 32 kB, 8-way associative VIPT caches, one for

the data and one for the instructions.

The design goal of the cache is to dynamically store values to reduce latency by preventing DRAM calls. To do so, it must store new data constantly and evict previously stored data. When new data is loaded in the cache, the cache uses a *replacement policy* to determine which line must be evicted. An intuitive replacement policy is Least Recently Used (LRU), where the data with the oldest last usage is evicted to store the new data. In practice, as this policy requires the hardware implementation of a timestamp for each cache load, processors used a pseudo-LRU policy with an approximation of last-used timestamps. However, this policy is suboptimal in some instances. For instance, when the user is browsing a congruent set of addresses larger than the cache associativity, all calls will result in a cache miss. Modern processors use more complex and hybrid solutions to adapt to various situations.

2.1.3.3. DRAM

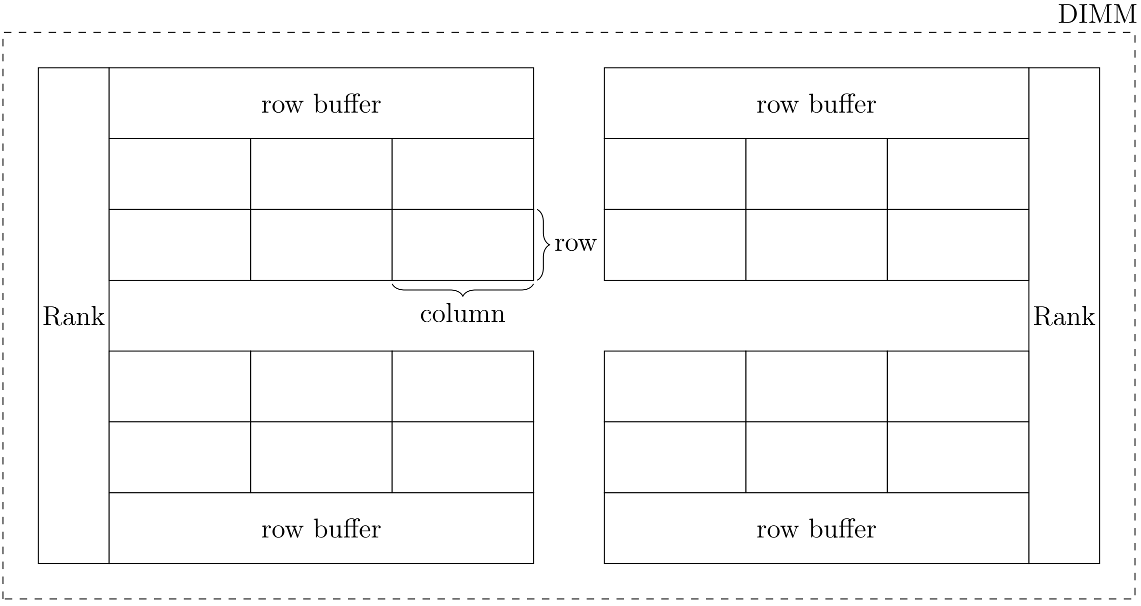


Figure 2.4. – Organization of a DRAM DIMM. It is composed of two ranks, each handling two banks. Each bank is composed of 3 columns and 2 rows.

Dynamic Random Access Memory (DRAM) is the main memory component in modern computer systems. Its access time is significantly higher than the cache's, but it can store more data. When loading an address, the processor first checks if the address is cached. If not, it then loads it from the DRAM. The processor handles DRAM with an on-chip memory controller. The memory controller communicates with the DRAM through one or more parallel channels. The DRAM is organized in Dual Inline Memory Modules (DIMMs). One or more DIMMs can be connected to the same channel. Figure 2.4 illustrates the organization of a DIMM. DIMMs are separated into one or two *ranks*, corresponding to the physical sides of the module. Each rank handles 16 *banks* on DDR4 and 32 on DDR5. The bank manages the actual memory, typically decomposed in 2^{17} *rows* and 2^{13} *columns*. The mapping of an address to channel, DIMMs, rank, and bank is not documented for Intel but has been

reversed by Seaborn et al. [Sea] on a Sandy Bridge processor and by Pessl et al. [PGM+16] in 2016 by exploiting a timing side channel. More recently, Wang et al. [WZCN20] introduced DRAMDig, a generic tool to reverse the address mapping of documented Intel processors.

When the processor wants to load a value from the DRAM, it looks for the corresponding channel, DIMM, rank, bank, and row. It then *opens* the row, i.e., copies the whole row to the bank's *row buffer*. The processor then reads the sought-after column from the row buffer. Repeatedly reading data from the same column always loads data from the row buffer without reloading it. However, reading data from the same bank but another row flushes the row buffer and loads the new row.

The DRAM's cells hold a volatile charge that decreases over time. When this charge gets under a threshold, the bit held switches from 1 to 0. Therefore, all the data in a cell is regularly refreshed to prevent data loss. This *refresh interval* is 32 ms in DDR4.

2.1.3.4. In-Flight Data

In-Flight Data are pieces of data temporarily stored in special buffers outside of the standard memory subsystem.

Store Buffers (SBs) are small buffers, connected to the store port and the L1 Data Cache. They aim at tracking pending stores. They implement the store-to-load optimization: the processor speculates if a load and a store have the same physical address before their translation from virtual address. If so, it will send data from the store buffer to the load buffer.

The Line-Fill Buffer (LFB) is a small buffer (typically 10 entries) handling outstanding memory accesses. CPU can speculatively load data from the LFB when it is not found in the L1 cache.

2.1.4. Hardware Performance Counters

Hardware Performance Counters (HPC) are special registers built in the micro-architecture of modern CPUs. They collect information about microarchitectural events at runtime, and are accessible from the software [Wikb]. They monitor various events, e.g., cache hits, misses, μops dispatched by ports. Initially used for debugging purposes, they can be used to evaluate the behavior of the microarchitecture at runtime.

2.2. Web Browsers

This section provides the necessary background on modern browsers' architecture to understand the attacks described in this manuscript. This thesis mainly focuses on two different browsers: Google Chrome and Mozilla Firefox. To that extent, we will provide more depth about these two browsers, but some generic aspects of browser architecture or microarchitectural attacks may be implementable on more browsers with some fine tuning. Furthemore, many browsers are forks of Chrome (e.g., Brave, Opera, or Microsoft Edge) or Firefox (e.g., ToR Browser) and architectural design or components remain the same. Firefox and Chrome represent 75% of the market share of dekstop browser. If we count all Firefox-based and Chrome-based browsers, this marker share reaches 91%.

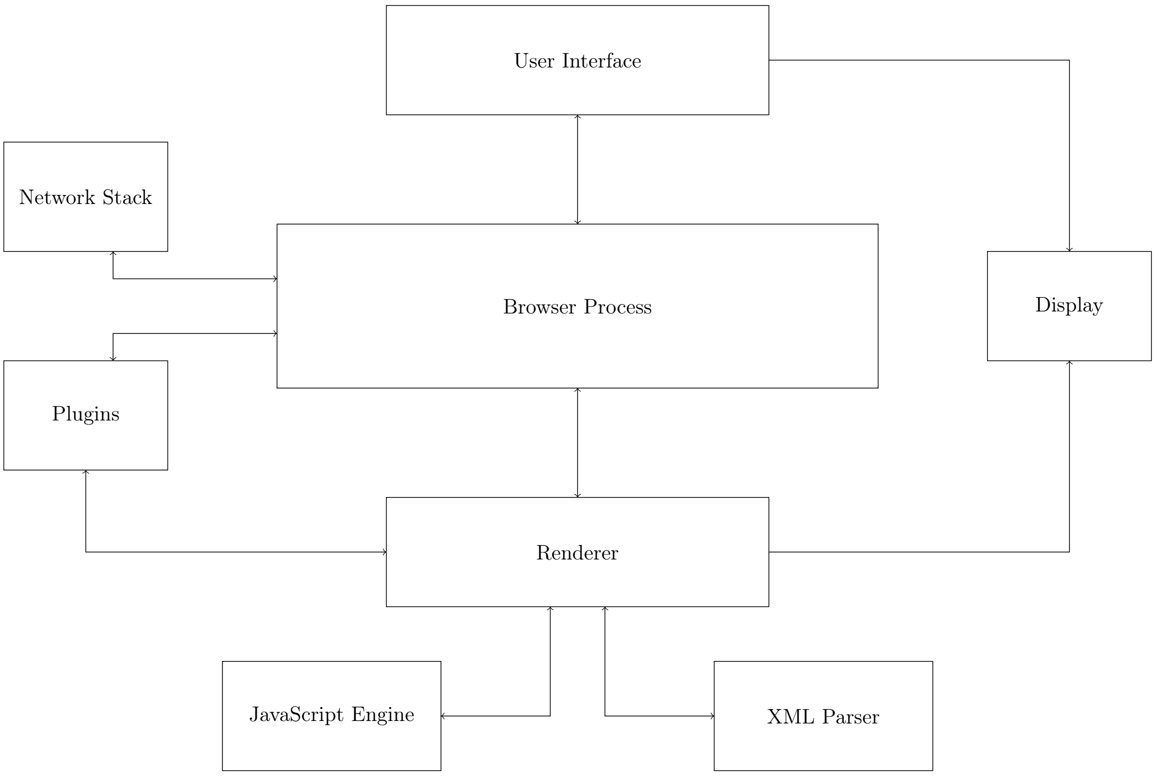


Figure 2.5. – Simplification of the structure of a modern web browser

2.2.1. Browser Architecture

Browsers are complex software composed of many different processes. Although Chrome and Firefox vary significantly in architecture, they have a similar high-level architecture illustrated in Figure 2.5.

Browser process: The browser process, or browser engine, is the central component of the web browser. Its main task is to allow communication between all components and synchronization and handle privileged options such as local file accesses. It is directly communicating with the User Interface (UI).

User Interface: The UI encompasses all the browser's components presented to the user, e.g., the URL bar, previous and next buttons, or the settings menu.

Network stack: The network stack is responsible for fetching the URLs provided by the user and forwarding the result to the browser process.

Rendering engine: The rendering engine, or renderer, controls the actual webpage in the browser. It is responsible for parsing the HTML/CSS code sent by the server and displaying it. The rendering engine parses all HTML code with the XML parser into a tree-like interface called Document Object Model (DOM). The Document node is the top node of the tree, and each element displayed on the page is a node below in the tree. In the meantime, the rendering engine parses the CSS attributes of the site,

essentially controlling the layout of the displayed page. Both the UI and the rendering engine are responsible for the display component of the browser, i.e., what the users see on their screen. The rendering engine is also responsible for parsing scripts in the page and sending them to the JavaScript engine. Chrome uses the Blink rendering engine, except on iOS versions where it uses WebKit. Firefox's rendering engine is called Gecko and is developed internally by Mozilla.

JavaScript engine: The JavaScript engine is the main component of client-side scripting on the web. It inputs client-side scripting code and compiles it to native machine code at runtime. Client-side code handles all interactions between the user and the webpage without communicating with the server. It typically handles forms, animations, modifications to the DOM, and generally makes the page dynamic. We will provide more details on client-side computing in Section 2.2.2.

As the webpage is an interactive environment, the renderer process is highly event-oriented. The browser must react to users' inputs. To handle asynchronicity, the renderer implements an *event loop*. The event loop is a FIFO queue, where each asynchronous event is registered. When browser resources are available, it will pop the first ready asynchronous event and execute the callback function. Modern renderer processes contain several event loops with different priorities, handling DOM events, network requests, or page repaints.

This multi-process architecture is also parallelized in the browsers. Each tab or origin runs its own renderer process, along with JavaScript engine, XML parser, or event loops. This allows more performant browsers, as processes cannot block each other's execution through the event loop. For instance, if a tab goes unresponsive because of heavy computations, the other tabs are not affected by the slowdown. It also grants more isolation between contexts. However, as some common infrastructure, e.g., JavaScript engine, are duplicated, having several processes can result in higher memory usage.

2.2.2. Client Side Languages

To enhance user experience with more dynamic pages, most websites encompass scripts aimed at being executed locally in the users' browsers. This provides faster computation, without the delay introduced by network computation, and reduces the servers' workload. JavaScript is the most used client-side scripting language, but other languages are used to fill specific needs. This section presents the necessary background on the principle and execution of JavaScript and WebAssembly, as they are the languages used in our attacks.

2.2.2.1. JavaScript

JavaScript is a lightweight, high-level scripting language. It is used as the client-side scripting language for most websites [w3t], but is also used in back-end programming with Node.js, managing databases, or as a framework to develop webpages. This section focuses on JavaScript's use as a web client-side language. JavaScript is defined by the ECMA standards [ECMc] as a single-threaded scripting language. However, due to its usage in web browsers, it has been enriched with APIs to manage multi-threading, DOM manipulation, animations, and many browser-specific features. Most of these applications are also standardized by the ECMA [ECMd]. For security reasons, JavaScript is executed in a secured environment, also

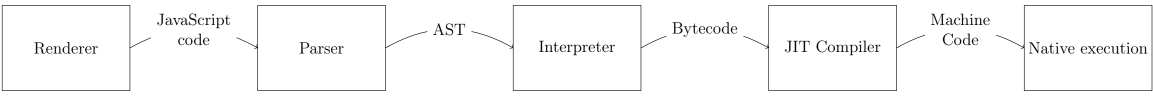


Figure 2.6. – JavaScript execution in the engine.

called a *sandbox*. JavaScript scripts have no access to virtual or physical addresses, native instructions, reads or writes on the filesystem, or to system information.

JavaScript engines use Just-In-Time compilation, i.e., they compile the code directly at runtime and not beforehand. The translation from JavaScript to machine code in the browser is handled by the JavaScript engine, e.g., v8 [Gooe] for Chrome and SpiderMonkey [Mozb] for Firefox. The translation from JavaScript to executable machine code is decomposed in several stages illustrated in Figure 2.6. The JavaScript is first parsed into an Abstract Syntax Tree (AST), i.e., a formal tree representation of the code. This AST is then used to generate bytecode, i.e., intermediary abstract code. At this point, the code is still portable, but the bytecode is loaded in memory and specific to a JavaScript engine. Both v8 and SpiderMonkey use a stack-like register machine [Mozc, Goof] to be as close to actual machine code as possible. Chrome's v8 use a parser and interpreter called Ignition [vba].

This bytecode is then compiled just-in-time to executable machine code. The JIT compiler introduces many different optimizations to the code, e.g., redundancy elimination, inlining, caching, or speculating on operand types. Firefox's JIT compiler is known as Warpmonkey [Mozb], and Chrome's is known as TurboFan [vbc].

2.2.2.2. WebAssembly

WebAssembly is a portable binary-code language standardized by the W3C [W3C]. It is designed to be used as a client-side language of the web, in addition to JavaScript. It is a bytecode-like language aimed at being run in a portable stack virtual machine to offer better performance than JavaScript. WebAssembly can be compiled directly from other languages, e.g., C or Rust, or written in the `wat` text format, an assembly-like S-expression representation of the binary code. It is built as a stack machine and has its own memory pages. To this day, WebAssembly supports more than 200 standardized instructions, including algebraic operations, memory management, and SIMD operations. WebAssembly is executed in the JavaScript sandbox and suffers from the same security restrictions.

WebAssembly is also compiled by the browser's JavaScript engine. However, WebAssembly being a binary language, the source files are already bytecodes. Thus, instead of being interpreted, they are directly compiled. Chrome's v8 implements a two-tiered compiler for WebAssembly illustrated in Figure 2.7. It is composed of Liftoff [vbb], a simple one-pass compiler. Liftoff has the advantage of swiftly compiling the WebAssembly bytecodes to reduce pages' startup time. TurboFan is also used to compile WebAssembly to machine code. It is a slower compiler but brings significant performance optimizations. When a WebAssembly script is loaded on a page, it is first quickly compiled by Liftoff, while the slower TurboFan compiles a more optimized version of the code. When the second compilation is done, the browser executes the optimized version. Firefox's SpiderMonkey uses a similar two-tiered compilation for WebAssembly, using a fast, one-pass compiler called RabaldrMonkey [Moza] and an optimized compiler, BaldrMonkey.

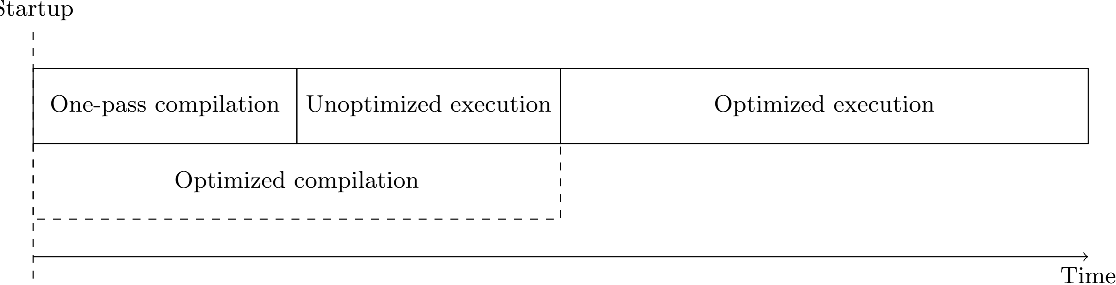


Figure 2.7. – Illustration of WebAssembly two-tiered compilation: the one-pass compiler compiles WebAssembly swiftly to reduce startup time, then when the optimized compilation has finished, the execution switches to the optimized compiled code.

2.3. High-Resolution Timers

Side-channel attacks often rely on small timing differences to infer sensitive data. A common prerequisite for these attacks is to be able to distinguish events in the order of the nanosecond. To do so, the attacker needs to have access to high-resolution timers. This section provides an overview of high-resolution timers, particularly in the restricted JavaScript sandbox.

2.3.1. Definition

In this manuscript, we call a timer, or a clock, a tool that differentiates two events based on their respective timings. To do so, a timer relies on an operation that needs to:

- Be constant over time to provide a reliable non-varying unit of time.
- Be free running to allow the computation of time differences without blocking the program execution.

We call clock edge the moment where a constant time operation ends and the next starts. A timer can differentiate two events if each crosses a different number of clock edges. The duration of the constant time operation is the smallest amount of time this timer can measure, i.e., its resolution.

2.3.2. High-resolution Timers in Native Environments

Intel x86 processors propose the TIME STAMP COUNTER, a 64-bit register counting the number of cycles since its last reset. The x86 `rdtsc` [Intd] instruction allows an unprivileged user to read and return the value of the TIME STAMP COUNTER.

This timer is *cycle accurate*, meaning it has a resolution of a single CPU cycle. It is the smallest time unit available for the processor. However, cycles are, by design, dependent on the CPU frequency. For instance, power-saving policies can reduce the processor frequency when the workload is low, reducing `rdtsc` resolution.

Out-of-order execution can also introduce significant overhead or randomness to the measurement. In particular, to measure a specific operation, we call `rdtsc` before and after

executing the operation. The difference between the two timestamps will correspond to the execution time of our operation in cycles. However, because of out-of-order execution, the processor will often execute both calls to `rdtsc` one after the other to boost performances. In that case, the difference in the timestamps is not the execution time of our operation. Adding MFENCE before and after each call to `rdtsc` prevents re-ordering [Intb].

2.3.3. High-resolution Timers in Web Browsers

Client-side languages of the web are executed in a heavily-restricted sandbox. This sandbox prevents access to native x86 instructions. Thus, attackers sitting inside of the JavaScript sandbox do not have access to `rdtsc` or other native cycle-accurate timers. JavaScript offers access to built-in timing sources, but academics have also leveraged other functionalities to create *auxiliary timers*.

performance.now() The JavaScript High-Resolution Time API [Gri] offers access to the `performance.now()` method, which returns a high-resolution timestamp. The timestamp value represents the time elapsed since the beginning of the current document lifetime in ms, initially with a resolution in the order of 1 ns.

For security reasons, `performance.now()`'s resolution has evolved over time. However, in 2017, Schwarz et al. [SMGM17] demonstrated that it is still possible to recover high-resolution timers regardless of the base resolution by using clock interpolation. The idea behind it is straightforward: one counts the number of times a shorter non-free running operation can be executed. We refer to such an operation as a tick. This tick can simply be writing repeatedly some data to memory or increasing a custom counter by one. In this manuscript, we use a simple JavaScript variable increment.

(a) The interpolated time is 4 ticks. **(b)** The interpolated time is 3 ticks.

Figure 2.8. – Clock interpolation: Counting the number of ticks between the end of the event and the end of the clock period. Even if both events are shorter than the clock period, we can distinguish them by their interpolated time: event 1 has an interpolated time of 4 ticks, whereas event 2 has only 3 ticks.

Figure 2.8 provides an example of how timer interpolation works. Events 1 and 2 have a shorter execution time than the clock period i.e., the resolution. Interpolation is then needed to differentiate them based on their timing. By running events at the beginning of a clock period and counting ticks when they are finished, we can conclude how fast each of them is when the next clock edge is reached. The more ticks are counted, the faster the event is. In Figure 2.8, Event 1 is faster than Event 2 as it has 4 ticks against 3. It should be noted that the interpolated timer is equivalent to a timer with a resolution equal to the duration of a tick.

SharedArrayBuffer Initially a single-threaded language, JavaScript has evolved into a multi-threaded paradigm. ECMA2017 introduced the `SharedArrayBuffer` API [ECMb] to accelerate computations between threads. It allows creating an array shared between the main thread and a sub-thread, or web worker. First implemented in Firefox 46 and Chrome 60, Schwarz et al. [SMGM17] used them to build a high-resolution timer. The generic idea of using shared array to build clocks has also been exploited natively when high-resolution timers were not available [LGS+16, LHS+20].

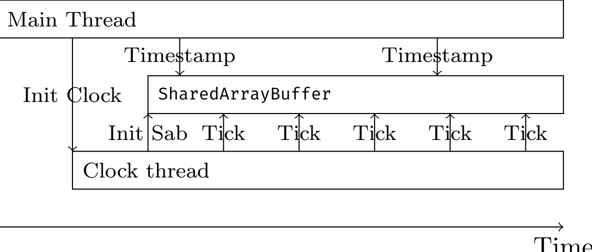


Figure 2.9. – Simple `SharedArrayBuffer` based clock: To time an event, the main thread evaluates the shared value before and after its event.

Figure 2.9 illustrates a simple `SharedArrayBuffer`-based clock. The attacker creates a *clock* web worker and shares a `SharedArrayBuffer` between the threads. The clock thread is an infinite loop, perpetually incrementing a value in the `SharedArrayBuffer`. This value can be consulted at any time by the attacker and represents a timestamp. The resolution of this timer is very high as it is of the order of the computation time of a shared operation, such as incrementing or reading a shared variable.

Other sources of timing The W3C standards describe other API timers accessible in JavaScript, such as `Date.now()` [Cond], `Window.requestAnimationFrame()` [Coni] or `Window.setTimeout()` [Conj], but they offer a resolution lower than the one of `performance.now()`. Schwarz et al. [SMGM17] also designed other auxiliary timers. They presented clocks based on CSS animations. An attacker can run JavaScript code at each screen refresh, i.e., every 16.66 ms for a typical refresh rate of 60 Hz, thus this can be used for interpolation as well. This is equivalent to using `Window.requestAnimationFrame()` directly in JavaScript.

2.4. Microarchitectural Attacks

In this section, we present an overview of microarchitectural attacks. They exploit the microarchitecture's optimizations to pass through software security barriers. We present three types of attacks:

Side Channels: the attacker exploits timing differences caused by the microarchitectural state.

Fault Attacks: the attacker actively injects faulty values in the microarchitecture.

Transient Execution Attacks: the attacker exploits instructions that are computed but never actually committed to the architecture.

Most of these attacks share a common threat model, where the attacker is running code on the victim's hardware. This can be an unprivileged native process or an attacker sitting in the JavaScript sandbox. Microarchitectural attacks exploit shared resources between legitimate and malicious processes.

2.4.1. Microarchitectural Side Channels

This subsection describes microarchitectural side channels. They leverage differences in hardware behavior caused by microarchitectural optimizations to leak sensitive information. They can be modeled as *channels* between two entities, similarly to network communication. An attacker can exploit these microarchitectural channels to communicate between two attacker-controlled entities with a *covert channel*. If an attacker creates a channel with a victim-controlled entity, leaking data unknowingly, this channel is a *side channel* attack.

Co-residency between the attacker and the victim is a significant part of the threat model. An attacker sitting on the same core as the victim can exploit on-core resources, e.g., CPU ports, to mount attacks, whereas an attacker sitting on a different core is restricted to cross-core components, e.g., the ring interconnect. Microarchitectural attacks can also be mounted cross-CPU, targeting components shared between processors, i.e., DRAM for large cloud environment.

2.4.1.1. Cache attacks

Cache attacks are a group of timing attacks exploiting the internal state of the CPU caches to leak information. The attacker modifies and monitors the cache state to spy on the victim process or create a covert channel. Cache attacks have been greatly explored, and several different attack primitives exist. As L1 and L2 caches are split between cores and the last-level cache is shared between all cores, cache attacks include on-core, cross-core and even cross-CPU attacks.

In native environments Hu [Hu92] mentioned for the first time the possibility of exploiting the cache state to mount attacks. They built a covert channel, allowing internal communication in the VAX security kernel. Cache attacks have largely been used to attack cryptographic implementations. The idea of exploiting timing differences resulting from the cache state was mentioned by Kocher [Koc96] and Kelsey et al. [KSWH00]. They analyzed various side-channel techniques on S-Box cryptography and highlighted the potential of attacks based on the cache hit ratio. This attack scenario was applied by Bernstein [Ber05] on AES. By exploiting how secrets were used as indexes in arrays, they recovered secret keys based on AES design and not exploiting a specific implementation. Osvik et al. [OST06] standardized the approach of cache attacks. Their attacks were mounted on real-world implementations of AES, such as OpenSSL or Linux's `dm-crypt`. By defining attack primitives, such as Prime+Probe or Evict+Time, they paved the way for a systematized approach to cache timing attacks. We will now describe the most common cache-attack primitives.

In Evict+Time [OST06], the attacker evicts a specific cache set s and then times the execution of the attacked function. If this execution time is longer than usual, then the function accessed data stored in s. This attack has a major prerequisite: the attacker must be able to repeatedly trigger the attacked function. It has a resolution of a single cache set. Osvik et al. [OST06] have been the first to exploit Evict+Time in an attack against OpenSSL's

AES device encryption in 500 000 samples, running in half a minute. Hund et al. [HWH13] leveraged Evict+Time to circumvent kernel space ASLR. By setting privileged code portions in the cache using system calls, accessing a set of user-space designated addresses, then rerunning the system calls, they can gather timing differences created by the cache state. If the second execution of the system calls is longer than the first, then the user-space addresses have evicted the privileged addresses. Due to cache indexation being based on addresses, this difference in timing leaks information on the physical or virtual addresses of the system-call addresses. Evict+Time has also been proven applicable to other microarchitectures. In particular, Spreitzer and Plos [SP13] mounted an attack on unaligned AES T-tables, managing to recover the private key in the first AES round. This attack was applied to ARM microarchitectures, particularly on mobile devices. It was mounted from an unrooted user on a Google Nexus S smartphone.

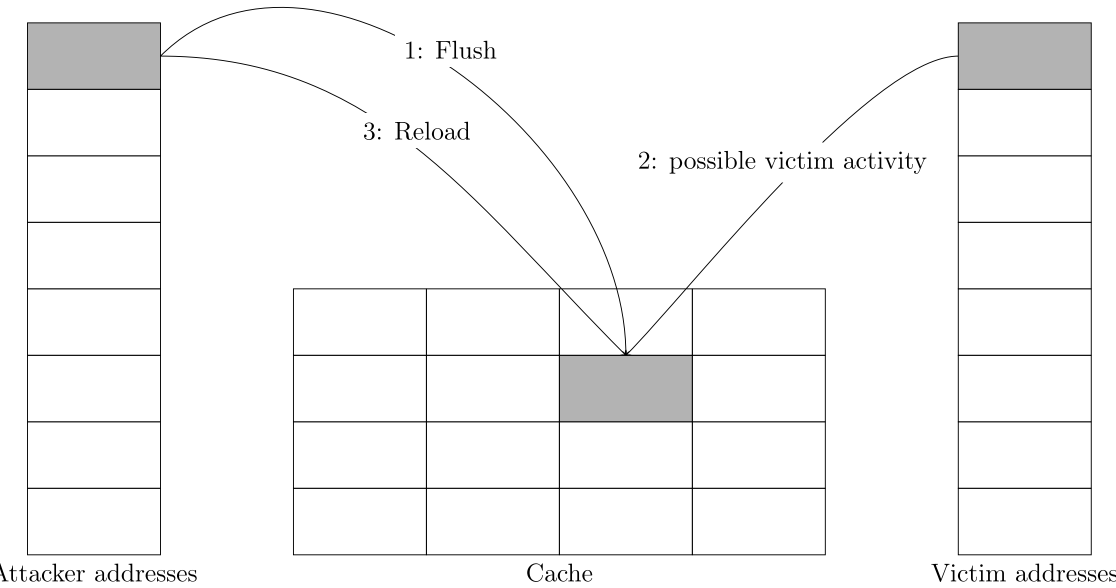


Figure 2.10. – Flush+Reload attack.

In Flush+Reload [YF14], an attacker can monitor a single cache line. Figure 2.10 illustrates the workflow of the attack. The attacker regularly flushes data from a specific cache line with the native `clflush` instruction and measures the time it takes to reload the same data after a short time. As Intel's caches are inclusive, the data is also flushed from the L2 and L1 caches. If the reload time is short, i.e., a cache hit, the data has been reloaded in the cache by the victim. If it is long, i.e., a cache miss, the data is not in the cache, thus, the victim process did not access it. This attack offers a single cache line spatial resolution and shows the best accuracy of existing cache attack primitives. It, however, has three major prerequisites:

- Access to the native instruction `clflush`.
- Inclusive caches.

- Shared data between the attacker and the victim.

The first attack in the Flush+Reload category was proposed by Gullash et al. [GBK11] in 2011. They recovered OpenSSL's AES secret keys in under 100 encryptions. Yarom and Falkner [YF14] standardized Flush+Reload and used it to mount an attack against RSA. Flush+Reload has been leveraged on attacks against AES implementations [GBK11, IIES14, AIES15, AES15, GIA+15, GSM15, AAG17, SS20], DSA [YB14, BvdPSY14, vdPSY15, ANT+20] or PAKE [BFS20, BFS21]. Flush+Reload attacks have also been applied to other microarchitectures, most notably ARM smartphones [LGS+16] in a variant named Evict+Reload. In this attack, the Flush phase is replaced by a cache eviction. By filling a cache set with its own value, the attack can remove other addresses in the cache, effectively equivalent to using `clflush` when the instruction is not available. Flush+Flush [GMWM16] is a variant of Flush+Reload where the Reload phase is replaced by another call to `clflush`. By measuring the execution time of flushes, the attacker can determine whether new addresses have been set in the cache.

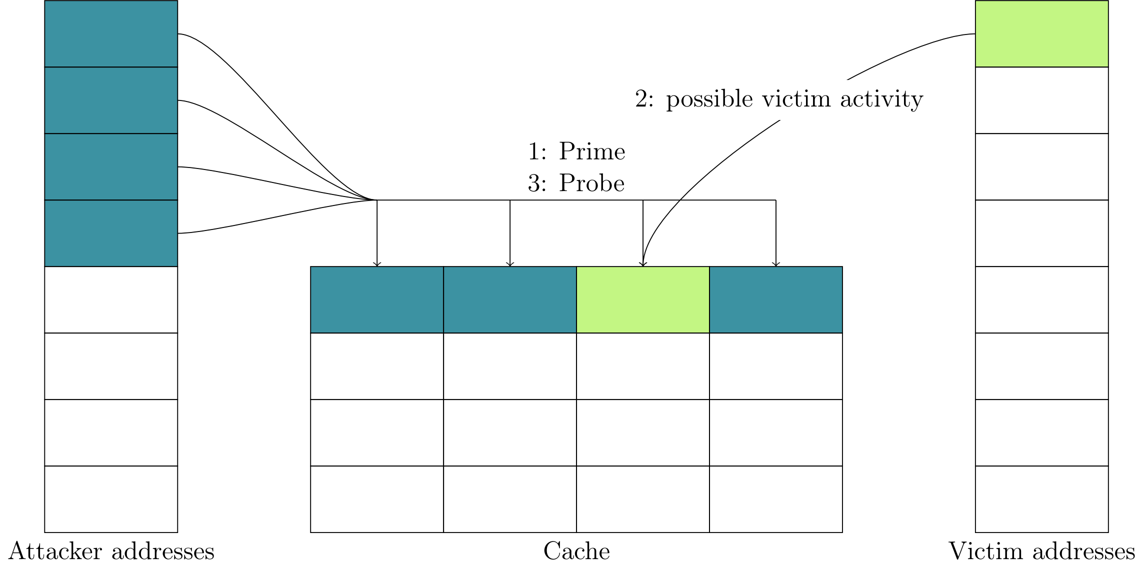


Figure 2.11. – Prime+Probe attack.

In Prime+Probe [OST06], the attacker first fills a cache set s with its own data (*Prime*). Then after a short time, it times the load of its data (*Probe*). If the reload time is short, i.e., only cache hits, it means that all the attacker data are still standing in the cache, thus the victim process has not loaded data stored in s. If the reload time is long, some attacker data have been evicted from s, i.e., the victim has loaded data stored in the monitored cache set. This attack offers a spatial resolution of a cache set, lower than Flush+Reload. However, the requirements are less constraining as the attack does not require access to native instructions or shared memory, allowing its usage in systems where an attacker cannot mount Flush+Reload. The main prerequisite for this attack is to build an eviction set, i.e., a set of data congruent to the victim, able to fill a specific cache set. Prime+Probe initially focused on the L1 cache, as building an L1 eviction set is trivial. As the L1 is virtually indexed, the attacker just selects addresses with the same index bits in the virtual address. Percival [Per05] was

the first to leverage it in an attack against RSA. L1 cache has been target by attacks against AES [OST06, NS06, OST06, BEPW10] or digital signatures [BH09, ABG10].

Creating an eviction set for the last-level cache is not a trivial task. As the last-level cache is indexed on the physical address, the attacker cannot determine the index bits of addresses. Liu et al. [LYG+15] proposed an algorithm to empirically create a minimal eviction set for an arbitrary cache set that is oblivious to memory addresses, thus usable in the JavaScript sandbox. For a cache with w-way associativity, we need at least w congruent addresses to evict a cache set. The algorithm is detailed in algorithm 1.

Algorithm 1: Naive algorithm to find an eviction set for a specific address.

```
1  Let A be a buffer in virtual memory, covering all the cache
     lines;
2  Let x be the address we want to evict;
3  Let τ be the threshold to distinguish a cache hit from a
     cache miss;
4  while | A |> w do
5  |   Access all the values in A to evict the whole cache;
6  |   Access x to set it in the cache;
7  |   Let y be a value selected at random in A. Access A\y ;
8  |   Let t be the access latency for x;
9  |   if t < τ then
10 |   |   The call was a cache hit and x was not evicted from
   |   |     the cache → y was a part of x eviction set;
11 |   else
12 |   |   The call was a cache miss, x was evicted → y was
   |   |     not a part of x eviction set;
13 |   |   Let A = A\y ;
14 |   end
15 end
16 Return A;
```

This algorithm creates a minimal eviction set for an address x by reducing a large, cache-filling buffer A. It removes a randomly selected address y at each step, then tries to evict x by browsing the set. If x is evicted, y is not necessary to build a minimal eviction set and can be safely removed. If x is not evicted, y is necessary to build a minimal eviction set. It is added back to A. We repeat this process until we reach $\mid A \mid= w$. It has a complexity of $O(\mid A \mid^2)$ memory accesses and $O(\mid A \mid)$ time measurements.

Vila et al. [VKM19] proposed an enhancement illustrated in algorithm 2. Based on group tests [Dam06], it separates the initial set A in $w + 1$ subsets. Instead of testing a single address at a time, it tests one of the subsets. At each set separation, there are w values in the final eviction set and $w + 1$ subsets, so at least one of the subsets does not contain a significant value. At every cache miss, $\frac{1}{w+1}$ of A is removed. This algorithm has a complexity of $O(w^2 \mid A \mid^2)$ memory accesses and $O(w^2)$ time measurements.

Algorithm 2: Group testing reduction for finding eviction sets on a cache with w-way associativity.

```
1  Let A be a buffer in virtual memory, covering all the cache
     lines;
2  Let x be the address we want to evict;
3  Let τ be the threshold to distinguish a cache hit from a
     cache miss;
4  while | A |> w do
5  |   Divide A in w + 1 size-equivalent subsets;
6  |   let C be one of the subset;
7  |   Evict the whole cache;
8  |   Access x to set it in the cache;
9  |   Access A\C ;
10 |   Let t be the access latency for x;
11 |   if t > τ then
12 |   |   The call was a cache miss, x was evicted from the
   |   |     cache. C did not contain any of the addresses in
   |   |     the eviction set;
13 |   |   Let A = A\C ;
14 |   else
15 |   |   The call was a cache hit, x was not evicted: C
   |   |     contains at least one address of the eviction set;
16 |   end
17 end
18 Return A;
```

With the introduction of an algorithm to build an eviction set for the last-level cache, new attacks exploiting Prime+Probe appeared, targeting the last-level cache. It allows to mount cross-core attacks in environments where Flush+Reload is not implementable, e.g., cloud environments. Zhang et al. [ZJOR11] exploited Prime+Probe in the cloud to monitor other containers running on the same hardware. Maurice et al. [MWS+17] used Prime+Probe to mount a 45 kB/s covert channel on Amazon EC2 cloud. It also has applications in secured enclaves such as Intel SGX [SWG+17, BMD+17].

Another critical factor of eviction-based cache attacks in the browser is the need to evict the set reliably, i.e., to evict the whole set rapidly and with a high probability. To do so, the attacker needs to consider the cache replacement policy. For a w-way associative cache, we have an eviction set $A = \{a_1, \ldots, a_n\}$. In a simple LRU policy, linear access $a_1, a_2, \ldots, a_n$ with $n = w$ is sufficient to reliably evict the whole set as each access to an address of A evicts a single address in the cache set. However, modern caches use more complex hybrid replacement policies, and a simple linear access pattern may not be sufficient. The attacker needs to access the eviction set in an optimized way for a specific replacement policy. We refer to this access pattern as an *eviction strategy*. Optimal eviction strategies vary with processor generation, cache associativity, or size. Gruss et al. [GMM16] propose a heuristic to find fast and reliable eviction strategies in JavaScript. They propose three parameters to modelize eviction strategies. The first is the number of repetitions for each address. For instance, accessing $a_1, a_1, a_2, a_2, \ldots, a_n, a_n$ is on average 33% faster than a simple linear access on a Haswell processor. The second parameter represents a sliding window on the eviction

set to repeatedly access subsequences of the eviction set, e.g., $a_1, a_2, a_1, a_2, a_3, a_4, \ldots, a_n$. Finally, the last parameter represents the step/overlap of the sliding window. By varying these parameters, Gruss et al. were able to get a 99.9% eviction rate with a faster eviction time than simple linear access on a Haswell i7-4790. An attacker can find the best eviction strategy to mount Prime+Probe attacks by empirically determining the best set of parameters for a particular system. However, this computation can be time-consuming, and Gruss et al. [GMM16] decompose it into an offline phase, where the best strategies for most systems are determined, and an online phase where the attacker tests previously computed strategies to isolate the most suitable.

Other cache attacks variants have been proposed to fit different threat models. Irazoqui et al. [IES16] propose Invalidate+Transfer, a new cache attack primitive working in a cross-CPU setting. It targets the coherency directory protocol of CPU interconnects. The attacker and victim sit on two different CPUs with different cache hierarchies but share a memory block. In the Invalidate phase, the attacker invalidates a block in their hierarchy, e.g., by flushing it. The directory protocol of the interconnect sends a message to other CPU's caches sharing this block to uncache it. Then, in the Transfer step, the attacker reaccesses the memory block. If another processor cached this block between the two steps, the directory is updated, resulting in a fast direct transfer access. If no other processors have accessed the block, the access is served from the DRAM, resulting in a longer access. This attack was leveraged to extract ElGamal keys on a shared AMD Opteron server. Prime+Abort is a variant of Prime+Probe introduced by Disselkoen et al. [DKPT17]. It replaces the timing phase of Prime+Probe by leveraging aborts from Intel hardware transactional memory implementation TSX, allowing to mount this attack on systems without access to high-resolution timers. Purnal et al. [PTV21] introduced Prime+Scope, a variant of Prime+Probe offering a higher time precision while having a similarly general threat model. The Prime phase is similar to Prime+Probe. Instead of probing the whole cache set for results, the attacker accesses the eviction candidate in the L1 cache. This prevents the observer effect, where the measurement of the attacker modifies the state of the victim's cache. For the attack to work, the attacker and the victim must share a last-level cache but not a L1 cache, i.e., sit on different cores.

In web environments Restrictions introduced by the JavaScript sandbox mitigate the implementations of some cache-attack primitives. In particular, the lack of access to native instructions and shared memory prevents the implementation of side channels based on Flush+Reload or Flush+Flush. Oren et al. [OKSK15] introduced the first cache-based side channel in the JavaScript sandbox. This attack is based on Prime+Probe and has been used to track the victim's browsing behavior. In particular, an attacker could monitor the cache state using Prime+Probe to determine which websites were browsed by the user in real-time.

Oren et al.'s attack uses Liu et al.'s algorithm [LYG+15], described in Algorithm 1 to build an eviction set for a target JavaScript variable. They proposed optimizations for this algorithm by considering the page structure. The addresses of the consecutive elements of the page are consecutive. Most modern processors have a cache line size of 64 bytes, meaning the offset requires $bit_{offset} = 6$ bits to encode. Knowing our array is composed of 4 kB, we can compare the last $bit_{page} = 12$ bits of addresses on a same page. This means we can guess the $bit_{page} - bit_{offset} = 6$ least significant bits of the cache index. Two addresses are congruent if they share the same index bits, so we can assume only one in 2^6 can be congruent with a particular set. Using this in the algorithm, we can only check the subset of possibly congruent

addresses, i.e., 1 in 64, speeding up considerably the algorithm.

Gras et al. [GRB+17] leveraged a JavaScript-based Evict+Time to fully break ASLR from a virtual environment. Genkin et al. [GPTY18] used a WebAssembly-based Prime+Probe to extract ElGamal, ECDH and RSA decryption keys from Google's End-to-End [Gooa] and OpenPGP.js [Ope] JavaScript cryptographic libraries. Shusterman et al. [SKH+19] define cache occupancy, a last-level-cache-wide Prime+Probe. By monitoring the usage of the whole last-level cache, they create memorygrams to identify which sites are being browsed by the user without building an eviction set. Shusterman et al. [SAO+21] mounted a Prime+Probe side channel entirely running in CSS and HTML, working in browser environments without JavaScript. They show how current browser restrictions, mainly based on JavaScript, can be circumvented to still fingerprint the website browsed by the users.

2.4.1.2. Cross-CPU attacks

Cross-CPU attacks target components that can be shared between different CPUs on the same system, e.g., DRAM. They mainly target multi-CPU environments, e.g., the cloud or portable Systems on a Chip (SoC), but can also be used in single-CPU environments in a cross-core setting.

DRAMA [PGM+16] is a side channel based on DRAM bank row buffer collisions. By exploiting the timing difference caused by row conflicts, the authors created a low-noise covert channel with a bandwidth of 2 MB/s. The receiver times the access to a specific address in the DRAM. To send a 1-bit, the sender accesses a different address in the same bank but in a different row. This causes a row conflict, thus a slower access time for the receiver. To send a 0-bit, the sender simply idles for a fixed time. DRAMA also proposed a side-channel attack. By identifying rows used by the victim in secret-dependent computations, the attacker can infer sensitive data. First, the attacker accesses an address in the targeted bank but in a different row than the victim's address. Then, it waits a short time for the victim to compute. Finally, it times the reaccess to its address. If the access is longer, it means a row conflict happened, i.e., the victim accessed an address in the targeted bank. This attack allows the attacker to monitor in real-time the victim's keystrokes.

The DRAM covert channels have also been exploited from the JavaScript sandbox. Schwarz et al. [SMGM17] implemented a covert channel based on DRAM row conflicts running from a native component to a JavaScript process. This sandboxed implemation brought challenges on fast cache eviction, as values served from the cache do not reach the DRAM and cannot cause conflict. Schwarz et al. also proposed a heuristic for address selection. JavaScript does not offer access to virtual or phyisical addresses. They exploited the on-demand heap allocation of JavaScript. These pages are backed by 2 MB Transparent Huge Pages. By browsing through a large array, the JavaScript receiver can detect a delay caused by a page change, indicating that the accessed address is at the start of a new page, i.e., the 21 least significant bits of the addresses are 0, granting information on the associated DRAM bank.

2.4.1.3. Cross-core attacks

Cross-core attacks target microarchitectural components shared between all cores, e.g., last-level cache, ring interconnect, or the DRAM. A cross-core attacker can target processes that do not achieve core co-residency.

Evtyushkin and Ponomarev [EP16] proposed a covert channel based on Random Number

Generation in Intel ISAs. It transmits information by creating contention on the Conditioner Buffer, which holds 4 64-bits random values. If values in this buffer are exhausted, it introduces a delay to regenerate them. The receiver times the *rdseed* execution. To send a 1, the sender exhausts values of the Conditioner Buffer, thus slowing down the receiver's *rdseed*. This cross-core covert channel shows a resolution of 100 kbit/s.

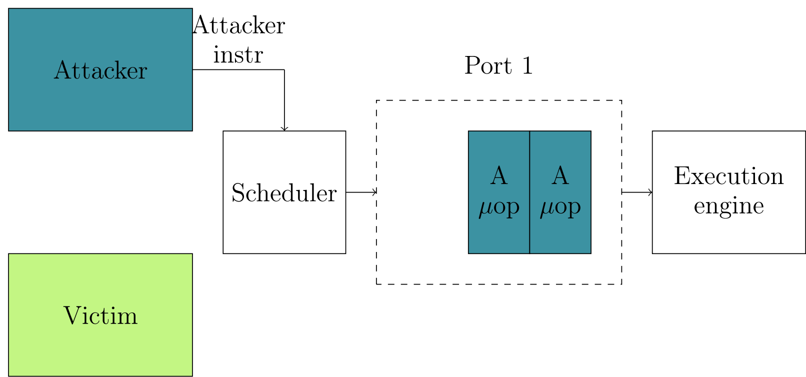
Paccagnella et al. [PLF21] proposed to mount side-channel attacks on the On-Chip Ring Interconnect. The attacker can create ring contention by "bombarding" specific ring regions with traffic. This contention causes differences in timing that can be used to mount a side channel attack against RSA implementations or to monitor keystrokes.

In the browser sandbox, Schwarz et al. [SLG19] automatically inferred microarchitectural information in JavaScript, such as the instruction-set architecture or the used memory allocator. Sanchez-Rola et al. [SSB18] exploited a timing side-channel based on the imperfections of processors' internal clocks. In particular, minor differences in the clock's quartz crystal can result in subtle timing differences for the same computations. By gathering execution times of cryptographic operations, they create unique hardware fingerprints for a machine.

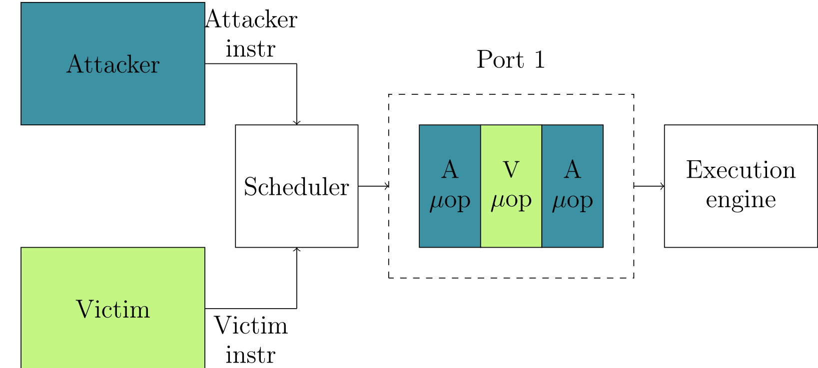
2.4.1.4. On-core attacks

On-core attacks target core-dependent microarchitectural optimizations, i.e., resources exploited only by processes running on a single physical core. On-core resources include the execution pipeline and its memory subsystem, including L1 and L2 caches. These attacks require the attacker and the victim to sit on the same physical core and thus are highly dependent on SMT.

Port contention side channels Aldaya et al. [ABuH+19] introduced PortSmash, the first port contention side-channel attack. Figure 2.12 illustrates how port contention can be leveraged in a side-channel attack. As a CPU port can handle a single μop per cycle, it can act as a bottleneck in the flow of operations. Thus, by repeatedly calling and timing instructions with a specific port usage, a spy process can monitor μops from other threads on the same physical core. For instance, an attacker can repeatedly call the `crc32` instruction, which is decomposed into a single P1 μop. This creates a bottleneck on P1. Next, by measuring the instruction's execution time, the attacker knows if instructions from other processes co-located on the same physical core are distributed on the same port. As illustrated in Figure 2.12a, if the attacker is the only source of μops on P1, all of its μops are executed in a row, resulting in a fast execution time. On the contrary, if the attacker's instruction has a longer execution time than usual, it means that another process has issued one or more μops to P1, as illustrated in Figure 2.12b. This difference in execution time allows an attacker to monitor other processes' port usage in real-time. Aldaya et al. were capable of creating and measuring contention on P1, P5 and P0156. Their side channel offers a spatial resolution of a single instruction. The attack is SMT-dependent, as the attacker and the victim must sit on the same physical core and share CPU ports. Aldaya et al. exploited this vulnerability to mount an end-to-end attack on OpenSSL's TLS implementation and recover private keys. In particular, they targeted the scalar multiplication code of OpenSSL 1.1.0h, in which port usage depends on the secret. Port contention was also leveraged by Aldaya et al. to leak data from an Intel SGX enclave.



(a) Victim has not used port 1: all attacker instructions are executed in a row.



(b) Victim emitted one μop on port 1: attacker instruction will be delayed.

Figure 2.12. – Illustration of port contention.

Other on-core sources of contention Other on-core resources have also been proven vulnerable to side channels. The Branch Target Buffer (BTB) has been exploited to leak cryptographical secrets [AKS07]. BTB mispredictions create a delay of a few cycles compared to an accurate prediction. This delay can be detected in the context of a cryptographic side-channel attack. Aciiçmez et al. propose several attack models. If the attacker and the victim achieve core co-residency, they share a common BTB. The attacker can then constantly clear the BTB, and cause a BTB miss during the execution of the target branch. Therefore, there will be a misprediction in the victim process when it takes the target branch. The attacker can then detect the delay caused by the misprediction to detect the branch taken and infer secret data. The BTB can also be exploited in a trace-driven approach. The attacker regularly executes branches BTB-congruent with the targeted branch. When the victim takes the targeted branch, one of the attacker branches is evicted from the BTB set resulting in a misprediction. By creating a trace with the execution times of all its branches, the attacker can detect mispredictions and create a real-time trace of the victim branch accesses. Evtyushkin et al. [EPA16] exploited a BTB side-channel to de-randomize virtual addresses in ASLR. By creating BTB set collisions between the attacker process and user-controlled kernel calls, the attacker creates timing differences, then can infer information about the kernel-space addresses.

Still in the Branch Prediction Unit, Evtyushkin et al. [ERAP18] showed that even a protected BPU implementation with a hardened BTB can fall victim to microarchitectural

side channels. In a new attack called BranchScope, they force a directional branch predictor to switch to a simple 1-level mode, letting an attacker communicate across user space or even from an SGX enclave. In concept, the attack is similar to a Prime+Probe but targeting the Pattern History Table (PHT). In a first step, the attacker sets the PHT in a desired state by executing a handpicked set of instruction. Then, after a short stalling time, they execute more branches targeting the same PHT entry. Based on the prediction outcome, the attacker is able to determine if another process has accessed the PHT entry between the two phases.

Translation Lookaside Buffers were also exploitable in side-channel attacks. In particular, Gras et al. [GRBG18] showed how, despite recent countermeasures on cache attacks, cryptographic implementations could be targeted by leakage caused by the TLB. The attack is similar to a Prime+Probe cache attack. The attacker creates an eviction set for a specific TLB set. By iterating through it, the attacker fills the TLB set with its own values, effectively evicting other addresses in the set. Then, after a short waiting time, the attacker measures the access time to addresses in the eviction set. If it is slower, then some addresses have been evicted, meaning the victim has accessed addresses stored in the targeted TLB set. Using this attack, Gras et al. leaked EdDSA and RSA secret keys in a single-trace setting.

The DSB, or μop cache, has also been exploited to mount covert channels [RMT+21]. By filling specific rows in the μop cache, the attacker can communicate and create a 900 kbit/s covert channel through different logical cores on the same physical core.

Attacks based on on-core contention have also been implemented inside the JavaScript sandbox. In particular, Andrysco et al. [AKM+15] leveraged a side channel based on floating-point operations to leak values of pixels rendered in the browser. By applying SVG filters with subnormal values and timing the rendering time, they were able to extract pixel values from the victim's browser. Stolen pixels could lead to monitoring or history sniffing.

2.4.1.5. Automated discovery of side channels

Finding target components for microarchitectural attacks requires heavy reverse-engineering and fine tuning to detect the vulnerabilities. Academics have proposed directions for more systematic and automatic approaches.

In a blog post [Fog], Fogh presented Covert Shotgun. Its goal is to detect possible on-core contention between pairs of instructions of an ISA. The framework executes each instruction of the pair on two logical cores sitting on the same physical core and measures their execution time. Instruction pairs presenting a longer execution time than usual reveal contention. The source of this contention is not precisely identified by the method but can be leveraged to create side-channel attacks or covert channels. Covert Shotgun was only tested on a limited subset of the whole x86 ISA but automatically found hundreds of possible instruction pairs creating contention, i.e., as many potential covert channels.

ABSynthe [GGK+20] follows a similar blackbox approach. It also aims at identifying potential on-core contention but targets the whole x86_64 ISA. For a pair of instruction (i_x, i_y), they automatically evaluate the impact of i_y on i_x's execution time. If i_x's execution time is longer when i_y is running on the same physical core, this reveals on-core contention. By repeating this experiment on all possible pairs of instructions, they create *leakage maps* of a specific microarchitecture, i.e., an exhaustive representation of contention caused by instruction interferences. Each pair of instructions creating contention represents a possible side channel, but the congested component is not identified. The authors then present ABSynthe, an automated side-channel synthesis. Given a specific microarchitecture (and its

leakage map) and a target process, it automatically creates a spy process that can, after a synchronized preprocessing phase, leak the target process's secrets without synchronization. ABSynthe is illustrated on an EdDSA implementation, effectively recovering 256-bit private keys in a single trace.

Osiris [WIN+21] expands side-channel synthesis to a larger scope than resource contention. They model side channels as a triplet of sequences:

Reset sequence: The attacker brings the targeted microarchitectural component into a known state, e.g., Prime of Prime+Probe or Flush from Flush+Reload.

Trigger sequence: The victim changes the microarchitectural state, e.g., modifying the cache state based on a secret.

Measurement sequence: With a timing measurement, the attacker determines if the target component is still in the reset state or has been modified in the trigger sequence. This is where the information leaks. Probe from Prime+Probe is an example of this phase.

For each triplet of instructions from a specific ISA, they determine if the measurements are similar with or without the trigger phase. Such triplets are potential side channels. Osiris rediscovered previously existing side channels, e.g., Flush+Reload, and also discovered 4 new side channels, enabling powerful attacks.

2.4.2. Microarchitectural Fault Attacks

Attackers can actively induce faults in the microarchitecture and exploit their effects to mount attacks. Hardware fault attacks have been largely studied, where faults are induced by changing the physical conditions, e.g., high or low temperature, laser, or radiation. Errors can also be entirely induced by software. In this subsection, we present an overview of software-based microarchitectural fault attacks.

2.4.2.1. RowHammer

In 2014, Kim et al. [KDK+14] proposed RowHammer, the first software-based microarchitectural fault attack. This attack exploits bugs in the DRAM. Reading values in the DRAM have side effects on physically adjacent rows. In particular, when a DRAM row is *hammered*, i.e., repeatedly accessed within a DRAM refresh window, the charge of physically adjacent rows may drop under a threshold, causing a bitflip from 1 to 0. This means an attacker can potentially modify bits in unauthorized memory areas. The rate of bitflip can increase if both rows adjacent to the victim are hammered.

Seaborn and Dullien [SD15] presented two different exploits leveraging RowHammer. The first is a sandbox escape from the Native Client (NaCl) sandbox. By causing bit flips in dynamically generated indirect jumps, the attacker can gain control over these jumps and access addresses outside of the sandbox. The second allows an unprivileged user to escalate to kernel privileges. Leveraging RowHammer, the attacker induces a specific bit in the Page Table Entry (PTE) to flip, making the PTE point to an attacker-controlled page. The attacker now has read/write access to its own page table, which effectively grants them control over the whole physical memory. Since these exploits, many RowHammer-based attacks have been developed, against cryptographic implementations [BM18, PSS+18], attacking or gaining access to cloud co-resident VMs [RGB+16, XZZT16], or triggering bitflips remotely [LAS+18, TKA+18].

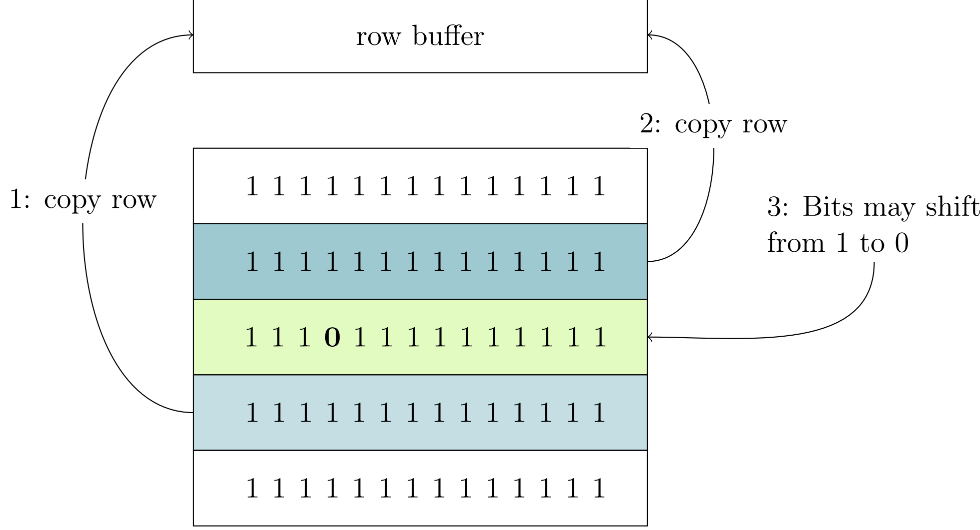


Figure 2.13. – Illustration of the RowHammer fault attack

Gruss et al. [GMM16] presented a RowHammer implementation running entirely in the JavaScript sandbox, allowing to remotely mount attacks from the victim's browser. Implementing RowHammer in JavaScript came with significant challenges. First, JavaScript is oblivious to virtual or physical addresses. The attacker does not know which address it is reading and thus does not know which DRAM row it is hammering. However, the attacker can exploit the 2 MB memory pages used by the browser. By iterating through a large array and measuring access latency, the attacker can detect page faults when the access time is significantly longer. This leaks information about the first address of the page. The attacker now knows that the following 2 MB represent 16 DRAM rows. It can cause bit flips by hammering rows at the edge of this page to cause bit flips in an adjacent, potentially sensitive page. Second, the attacker has no access to the `clflush` instruction. If the attacker repeatedly accesses the hammered address, it is served from the cache, not from the DRAM, thus preventing bit flips. The attacker has to ensure that the address does not sit in the cache without `clflush`. The authors propose *eviction-based* RowHammer. Similarly to Prime+Probe, the attacker composes an eviction set for the targeted address. By iterating through this set, they can ensure the eviction of the targeted address, which is then served from the DRAM. However, this eviction and DRAM load must have a high frequency to cause bit flips. The author proposed strategies to empirically determine optimal eviction strategies to maximize the eviction rate and minimize the eviction time.

2.4.2.2. DVFS-based fault attacks

Dynamic Voltage and Frequency Scaling (DVFS) is the energy management system of processors. It allows to dynamically lower processing speed to save energy. For instance, when the workload on a core is low, the DVFS reduces the core's frequency to reduce its energy consumption. When the workload increases, DVFS can raise the frequency to boost performances. DVFS can be controlled from the software, for instance with Linux's `CPUfreq` scaling governor.

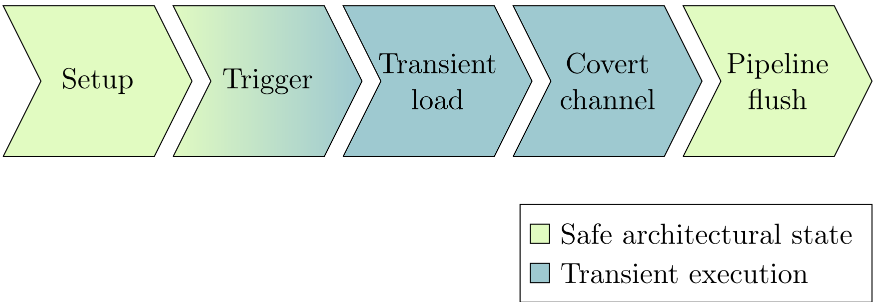


Figure 2.14. – Illustration of the workflow of transient execution attacks.

CLKscrew [TSS17] is a software-based microarchitectural fault attack exploiting DVFS. Flip-flops are memory components that change their value on the rising edge of the core clock. To locally propagate their output to adjacent flip-flops, the duration of the interval between two rising clock edges must be greater than a time interval defined by flip-flops' hardware properties. By overclocking i.e., over-extending the frequency at a specific time, a software-based attacker can induce bitflips by exploiting flip-flops' minimal timing constraint. The authors leveraged CLKscrew to extract secret keys or privilege-escalate in Trustzone in ARM/Android smartphones.

Plundervolt [MOG+20] is another software-based fault attack leveraging DVFS. They exploit the side effects of briefly lowering the voltage on the transistors, possibly creating faults. As opposed to CLKscrew, their attack focuses on x86 microarchitectures. They show the first occurrence of fault attacks on an Intel SGX enclave. Notably, an attacker leveraging Plundervolt can cause faults in the execution of hardware AES-NI computations in SGX, effectively allowing them to extract secret keys from the enclave in a few minutes.

2.4.3. Transient Execution Attacks

Transient instructions are instructions that often result from unauthorized computations, e.g., mispredictions or page faults. When detected, these faults cause a full pipeline flush and a roll-back of all operations. This flush resets the architecture at the previous state. However, the microarchitectural state may be modified by transient instructions' side effects. Mispredictions in speculative execution are a source of transient instructions. For instance, when the dependency is finally resolved and the wrong branch was taken, all μops from the mispredicted branch are purged from the ROB. Nevertheless, speculative execution is not the only source of transient instructions. Faults can also cause transient execution. When a page fault occurs, subsequent operations can still be processed in transient execution. They are later flushed but may also impact the microarchitectural state.

Transient execution attacks follow an execution pattern illustrated in Figure 2.14:

1. The attacker sets the microarchitecture in a specific state to prepare the transient execution and data extraction.

2. The attacker triggers transient execution, for instance by forcing a misprediction or forcing an instruction fault.

3. The transient instructions access the victim's sensitive data.

4. The attacker needs to extract the private data. As the data is flushed from the microarchitecture, the attacker cannot simply read it. Most transient execution attacks use a microarchitectural covert channel, where the transient instructions are the sending component.

5. Finally, transient instructions are flushed and the architectural state is reset, but the attacker can read the sensitive data from the microarchitectural covert channel.

Although the workflow is similar, transient execution attacks vary by the trigger phase, e.g., speculative execution or fault, on which address space the read happens, or the covert channel to extract data. Canella et al. [CBS+19] propose a classification of transient execution into two major classes: Spectre-type and Meltdown-type.

2.4.3.1. Spectre attack class

Spectre-type attacks exploit the transient window created by a control or data flow misprediction in speculative execution. Since the initial Spectre attack [KHF+19], many variants have been proposed, targeting different predictive components, poisoning strategies, or extraction covert channels. Canella et al. [CBS+19] propose a first classification based on the exploited victim component:

Spectre-PHT [KW18, KHF+19] targets the Pattern History Table (PHT), which predicts the results of conditional branches.

Spectre-BTB [KHF+19] exploits the Branch Target Buffer, which predicts the branch target for indirect branches.

Spectre-RSB [KKSA18, MR18] exploits the Return Stack Buffer responsible for return address predictions.

Spectre-STL [Hor18] exploits Store To Load data dependencies, where the memory disambiguator speculates on loads before all store operations on the same location have completed.

Another essential property of Spectre-like attacks is the poisoning method, or how the attacker "trains" the branch predictor to induce the misprediction in a particular branch in the victim execution. In particular, an attacker can exploit the virtual address indexation of branch prediction units to poison a branch predictor on a different address space in a cross-process setting, or directly train the branch predictor in the victim address space in a same-process setting. In native code on Intel processors, Spectre-PHT, Spectre-BTB, and Spectre-RSB all work both in same-process and cross-process settings, whereas Spectre-STL is only executable in same-process settings [CBS+19].

As cache side channels are the most studied in the literature, most Spectre-like attacks use cache-based covert channels to extract stolen data. In particular, the original Spectre and most of the following attacks leveraged Flush+Reload as their covert channel. However, Bhattacharya et al. [BSN+19] presented a Spectre-like attack exploiting port contention as a covert channel. By achieving core co-residency with the victim, the attacker can monitor its port usage with a high spatial resolution. This variant broadens the scope of potentially vulnerable code gadgets, as a single instruction can be leveraged to create detectable port contention.

Spectre attacks can be implemented in the JavaScript sandbox. In the original Spectre paper [KHF+19], Kocher et al. propose a JavaScript implementation of Spectre-PHT in the same-process setting. It allowed an attacker in Chrome 62 to read private memory in the browser's process by training the Pattern History Table to always predict a particular branch by passing it in-bound values. Then, on the last conditional branch, the attacker passes an out-of-bound value, resulting in a transient read to private data. The main difference with native Spectre attacks is the covert channel. As Flush+Reload is not implementable in the JavaScript sandbox, the authors leveraged Prime+Probe to extract sensitive values. Maisuradze and Rossow [MR18] presented ret2spec, a Spectre-RSB type attack that can also be implemented in WebAssembly, entirely running in the browser's sandbox. It also allows an attacker to arbitrarily read memory in the same-process setting. In 2021, Google's security team released a proof of concept of a same-process Spectre-PHT [Teab] in recent versions of Chrome, supposedly hardened against Spectre attacks.

2.4.3.2. Meltdown attack class

Meltdown-type attacks rely on the transient execution window after a CPU exception. These exceptions are handled by the architecture when the faulting instruction is retired, i.e., all of its μops are executed and previous instructions have been retired. When the faulting instruction is retired, the pipeline is flushed and the architectural state is set in a safe setting. However, between the faulting instruction emission and its retirement, an attacker can exploit the transient window where instructions following the faulting instructions are transiently executed on unauthorized data. These instructions will never be committed to the architecture, but an attacker can retrieve the information from the side-effects left on the microarchitecture by these transient instructions with a covert channel.

The Meltdown class has a wide variety of exploits and variants. Canella et al. [CBS+19] introduced a classification of Meltdown-like attacks based on the type of fault creating the transient execution. Many Meltdown variants exploit page faults [LSG+20, KW18, BMW+18] where the attacker tries to read unauthorized memory. In that setting, the system throws a page fault, but the data from the read is transiently forwarded to the following instructions in the pipeline, letting an attacker extract them. Applications to Meltdown-PF include leaking the entire kernel memory or reading values out of Intel SGX and generic hypervisors. Canella et al. [CSH+20] presented a page-fault-based Meltdown attacks running entirely in the JavaScript sandbox on 32-bits x86 Linux. Other exceptions were leveraged to mount Meltdown attacks, such as bound-range-exceed, where the attacker triggers the fault by accessing out-of-bound array indexes [CBS+19], general protection faults for unauthorized reads to privileged registers [Teaa], or lazy context switches in the FPU [SP18].

Van Schaik et al. [vSMÖ+19] presented RIDL, a Meltdown-type attack allowing to leak in-flight data from the Line Fill Buffer (LFB). The attacker first sets private values in the LFB, for instance with kernel calls. Then, by loading a new page, it forces the CPU to speculatively load a value from memory, which can be uncached in-flight private values loaded from the Line Fill Buffer. This incorrect load causes a fault, but the transient load is forwarded to transient instructions as in other Meltdown-type attacks. As the attack does not leverage physical or virtual addresses, it allows an address-oblivious attacker to leak privileged information. Such attacks are often referred to as Microarchitectural Data Sampling (MDS). RIDL attacks can leverage different faults to create the transient window, but always exploit the LFB as a source of leakage. RIDL was also implemented entirely in the JavaScript sandbox, where

the attacker is completely oblivious to physical or virtual memory addresses. It allowed an attacker in WebAssembly to leak victim data from a process running on the same physical core.

Schwarz et al. [SLM+19] presented ZombieLoad, another MDS attack targeting in-flight data, allowing to leak data from the LFB even if it is cached in the L1. ZombieLoad works on systems hardened against Meltdown attacks, such as Intel's CascadeLake generation, which is not affected by RIDL. The authors present various application scenarios for ZombieLoad. Notably, they showed how an unprivileged user can steal 128-bits AES-NI keys in under 10 s in a cross-process attack.

2.5. Side-Channel Attacks on Software and Browser Resources

This section presents an overview of side-channels attacks targeting software resources. These resources include *system-level resources*, managed by the operating system, and *application-level resources*, managed by the various programs. In particular, we take a strong interest in browser resources, as they are a core component of the contributions of this manuscript.

2.5.1. Attacks on system resources

Suzaki et al. [SIYA11] introduced an attack targetting memory deduplication. By creating a page in their address space with the exact same content as a victim memory page, the attacker possibly triggers the OS's memory deduplication. Then, the attacker writes on their own page. If the page was deduplicated, this triggers a page fault, resulting in a slower write. If the page was not deduplicated, the write access is fast. This difference in timing lets the attacker know whether or not their page was duplicated, i.e., that another process had a memory page with the exact same content. Suzaki leveraged this exploit to detect the presence of sshd or apache2 on the machine from inside a KVM virtualized environment. They also apply memory-deduplication attacks on images, showing how the same attacker, inside a virtual machine, can detect a file downloaded by a browser. Owens and Wang [OW11] showed how a virtualized attacker could exploit memory deduplication to fingerprint the OS. By reproducing OS specific memory pages, the attacker can detect when deduplication occurs, leaking information on which OS is installed from the virtual machine. Xiao et al. [XXHW13] used deduplication as a physical layer to create a 90 bit/s covert channel between two virtual machines running in the same system. Barresi et al. [BRPG15] introduced an exploit to fully reverse ASLR directly from a virtual machine. Schwarzl et al. [SKLG21] mounted fully remote memory-deduplication attacks based on HTTP requests, allowing to fingerprint server information or reverse KASLR over the internet.

Memory-deduplication attacks were also implemented in the JavaScript sandbox. Gruss et al. [GBM15] showed how a JavaScript attacker could detect running applications as well as monitor websites opened in other tabs in the same browser. The main challenge of a JavaScript-based memory-deduplication attack is to create exact copies of the victim's memory pages. To do so, they leverage JavaScript engine's implementations of `malloc`, creating page-aligned arrays. This allows an attacker to target page-aligned elements, e.g., CSS style sheets or images in browsers. Bosman et al. [BRBG16] showed how a JavaScript-based attacker could leverage page deduplication and the RowHammer bug to gain arbitrary read and write access in Microsoft Edge in a same-address-space setting.

Jana and Shmatikov [JS12] described a new memory-based side channel. By identifying processes' footprints on memory, an attacker, here a malicious application on a Android phone, can infer private information. Their attacker can infer which websites are being browsed by the victim in real-time, as well as the state of the browsing session. The authors also show how context switch timings can leak information on the user's keystrokes. A Linux-based attacker can monitor the `/proc/<pid>/schedstat` system file to retrieve the timing between two keystrokes. Even on Android systems, where this file is not accessible by unprivileged applications, the attacker can leak context switch information from the `/proc/<pid>/status` file instead.

Schwarz et al. [SLG+18b] showed how even an unprivileged sandboxed attacker can mount keystroke timing attacks by exploiting timing differences caused by system interrupts. The attacker repeatedly calls a high-resolution timer, e.g., `rdtsc`. If the OS throws an interrupt, the measurement process is interrupted, resulting in a significant cycle difference between two subsequent timestamps. As I/O operations, including keystrokes, cause system interruptions, an attacker can measure the time between these peaks to retrieve timing information on the user's keystrokes. This information can then be used, along with machine-learning algorithms, to reconstruct typed information such as a PIN codes or URLs [SWT01, ZW09]. Lipp et al. [LGS+17] implemented this attack entirely in the JavaScript sandbox. The main challenge of porting this particular attack to JavaScript is the lack of high-resolution timers. Instead of repeatedly calling a timestamp, the attacker measures the number of increments between two clock edges, similarly to clock interpolation. A drop in the increment count indicates the measurement has been interrupted. The attacker can use these drops to extract timing information about the victim's keystrokes.

2.5.2. Attacks on browser resources

Modern browsers are considerably large software that include many optimizations. In particular, components may be shared between origins, tabs, or contexts for optimization and browsing performance reasons. Sharing a component between the victim and the attacker can leak information about the victim's secrets or behavior.

Mowery et al. [MBYS11] showed how a JavaScript attacker can retrieve useful fingerprinting information about the browser vendor, version, or JIT engine by measuring the execution time of specific computations. They also show how an attacker can take advantage of the NoScript extension. NoScript is a browser extension that lets users block JavaScript from unknown origins. It is based on a whitelist system, i.e., the user defines which domains can run JavaScript code in the browser. By including many sources of codes from different origins on its own website, the attacker can identify which are whitelisted and which are not. This information can be used as a fingerprint of the user as the whitelist is custom.

Sanchez-Rola et al. [SSB17a] presented how differences in timings caused by access-control policies leak information about which extensions are installed. Browsers keep an access-control list to prevent third-parties, e.g., a JavaScript attacker, from accessing unauthorized extension resources. When a JavaScript attacker tries to access an extension resource, the browser typically first checks if the extension is installed, then checks the access-control list to see if the third party can access the extension. When they cannot access an extension's resource, a JavaScript attacker can measure this access request time to determine whether or not it was declined because of access control or because the extension is not installed. An attacker can then create a browser fingerprint by inferring which extensions are installed, even with no

permissions.

Van Goethem et al. [vGJN15] introduced several timing attacks against shared browser resources. In particular, they target the loading time of external resources, the parsing time of multimedia resources, the application cache, or the Service Worker Cache API to leak personal information in a cross-origin setting.

Vila and Köpf [VK17] exploited Chrome's shared event loop to monitor users, infer browsed websites, or create a covert channel. By creating contention on JavaScript's main event loop, the attacker can measure delay introduced by other processes using performance-heavy JavaScript computation. If the attacker fingerprints the impact of a specific website on the event loop, they can use it to recognize that website in an online setting. Similarly, monitoring the event loop lets the attacker infer the user's actions and idle times.

2.6. Countermeasures to Side Channels

This section presents an overview of side-channel countermeasures. Researchers have proposed a wide array of countermeasures, either specific to a particular vulnerability or more generic approaches. We present three primary paradigms of countermeasures: preventing the leak, preventing the measurement, and detecting the attack. As microarchitectural vulnerabilities are at the frontier of software and hardware, mitigations can be implemented at several levels: hardware, system, or application level, particularly in the browser.

2.6.1. Preventing the data leak

The most naive approach to preventing side channels is probably to fix the source of the data leakage. In particular, most side channels exploit the sharing of a component between the victim and the attacker to leak information.

2.6.1.1. Hardware Level

Architecture researchers have proposed several hardware-level mitigations to microarchitectural data leaks. Most of these solutions rely on resource partitioning, i.e., that the resources are not shared between the attacker and the victim.

Cache partitioning has been a widely studied mitigation to cache side channels. Static sharing [Pag05] is probably the most straightforward cache partition: the cache is split into multiple partitions, each assigned to a specific process. That way, data in a set cannot be evicted or flushed by malicious processes. Similar static sharing has been proposed at the logical thread level [OST06]. However, statically sharing the cache comes at a dramatic performance cost, as it reduces the cache capacity for each process and loses the shared memory optimizations brought by VIPT addressing. Dynamic partitioning introduces less performance overhead than static partitioning. Wang and Lee [WL07] proposed a Partition-Locked Cache, where users could activate a protection bit on cache lines, preventing them from being evicted by other processes. Vanguard [SLCO18] uses a different approach to cache partitioning: instead of assigning lines to a specific process, processes can only replace their own data in a cache set. This replacement policy modification prevents an attacker from evicting or flushing victim data while keeping the advantages of a shared cache. Other approaches have been proposed, e.g., partitioning at the set level [DJL+12], or runtime approaches based on hit/miss rates [WC14, WFZ+16].

Table 2.1. – Overview of side channels countermeasures.

	Preventing leakage	Preventing measurement	Detecting attack
Browser	• Same-Origin Policy [Cong] • Site Isolation [RMO19] • COOP/COEP [AJ]	• Removing high-resolution timers • Deterministic browsers [CCLW17]	
Application	• Secret-independent Code	• Blinding	
System	• Memory partitioning [KPM12, Inte] • Thread partitioning • Temporal partitioning [GZ13, VRS14, ZR13]	• Fuzzy Timers [VDS11]	
Hardware	• Cache partitionning [Pag05, OST06, WL07, DJL+12, WC14, WFZ+16] • Dynamic SMT [TP19, TRVT22]	• Randomized Cache [WL07, LL14, Qur18, THAC18]	• HPC-based detection [CSY16, AYQ+16, PIO19, LG18, For18, WSS+20]

Sharing components is also at the root of SMT-based attacks such as port contention. The most naive mitigation to these attacks is to simply disable SMT. However, this proposition represents a considerable performance degradation of up to 15% [BSN+19], as SMT allows for a highly efficient use of hardware resources. Townley and Ponomarev [TP19] proposed SMT-COP, a hybrid approach based on statically partitioning the use of resources between threads. This partitioning could be either temporal, with each thread accessing the resource after the other, or spatial, with each thread having its execution units. Their approach must be supported by the hardware and introduces a performance overhead of 8% compared to standard SMT, while preventing most contention-based side channels on the execution units or ports. More recently, Taram et al. proposed SecSMT [TRVT22], focusing on more secured shared resources against contention-based side channels. Their approach introduces, at the hardware level, different ways to share resources. In static partitioning, the resources are statically shared between logical cores. In dynamic partitioning, the partition of resources evolves according to the workload of logical cores to enhance parallelization. However, the

resources are never used by both cores simultaneously. More interestingly, asymmetric partitioning relies on different levels of trust. This model gains even more performance by letting a low-level security thread leak information to a high-security thread, but not letting high-security information leak to other threads. This is particularly interesting in a browser-based scenario. It is unsafe to leak information to the sandbox, whereas leaking sandboxed information to other threads presents fewer threats. Their asymmetric partitioning presents almost no overhead compared to traditional SMT.

Spectre-specific countermeasures were also proposed. Intel proposed to use logic fences in sensitive computations to prevent speculation [Inta]. The instruction sequence Return Trampoline (retpoline) was proposed by Google [Tur18] to mitigate Spectre-RSB. Placed into a source code, it forces mispeculation in a specific branch containing a fence, preventing speculation.

2.6.1.2. System Level

The OS can enforce partitioning at the memory level. Such approaches have been proposed against cache attacks in a cloud scenario. Kim et al. [KPM12] proposed to create stealth memory, i.e., areas of memory that are not set congruent with data from other processes. As the mapping of addresses to cache sets is deterministic, the system has to ensure that all pages congruent to a specific set are either locked or assigned to a single process. Processes can then set sensible data in stealth memory and ensure no data leak from the cache. Cache Allocation Technology [Inte], is an Intel software interface controlling the cache space allocation. It can operate at several granularities, restricting the cache space allowed to a thread, process, or VM.

The OS can use the scheduler to enforce spatial partitioning of resources with a thread granularity. For instance, allowing highly sensitive operations, such as computations depending on a secret, to run on a different physical core than other applications could reduce the risk of leaking private information in a side-channel attack depending on SMT such as port contention. Similarly, only sharing hardware resources between processes owned by the same user could provide more isolation, especially in cloud environments.

Temporal partitioning can also be enforced by the operating system. For instance, by flushing the cache [GZ13, VRS14] or internal buffers, e.g., the BTB [ZR13] at each context switch reduces the sharing of private data, thus reducing the risks of microarchitectural side channels.

2.6.1.3. Application Level

A generic application-level mitigation to microarchitectural attack is to ensure that the microarchitecture state is not dependent on secrets. In particular, cryptographic libraries try to ensure constant-time and cache-independent execution of computations using the secret key. Port-independent code has also been suggested [ABuH+19]. If the port usage does not vary according to the secret information, then port-contention-based side-channel attacks are ineffective. However, such a solution requires detecting and correcting all sensitive code in existing sensible implementations, and for all vulnerable microarchitectural components, and is not effective against covert channels.

2.6.1.4. Browser Level

Isolation is a staple of browser security. Separating resources and communication between contexts drastically reduces the threat surface of timing side channels.

The same-origin policy [Cong] is a central security feature of the web. An origin is defined by the tuple (Protocol, Port, Host). The same-origin policy restricts read access to resources loaded from a different origin. This means that data from one website, e.g., authentication cookies, are not accessible from another website loaded in another tab. This countermeasure aims at mitigating, among others, timing attacks using browser resources. However, the same-origin policy has no impact on other types of timing attacks. In particular, Spectre [KHF+19] was also implemented in JavaScript, therefore demonstrating the limitations of the same-origin policy.

In response to transient execution attacks, Chrome developed a new security measure called site isolation [RMO19], which forces each website, defined by the tuple (Protocol, Host), to run in a specific process, not shared with other websites. This prevents an attacker to access the mapped memory space of another website and mitigates the effect of process-space JavaScript-based transient execution attacks by preventing the access—including transient—to out-of-bound information. This feature was deployed in Chrome 67, and was deployed in Firefox 95 [Wika].

Site isolation was extended with the introduction of Cross-Origin Embedder Policy (COEP) and Cross-Origin Opener Policy (COOP) [AJ]. COOP ensures that a top-level window is isolated from other documents by putting them in a different browsing context group. For instance, if a website opens a pop-up whose origin is different from the website, the browser under COOP puts the pop-up in a separate process, similarly to site isolation, and prevents direct interaction with the main document. COEP complements COOP by forcing the browser to only load trusted resources. If a resource has no explicit permission to be loaded, the browser does nothing if COEP is enabled. Both COOP and COEP rely on policies defined through HTTP headers and they were both added in Firefox 79 and Chrome 83 [Conb, Cona]. When they are enabled on a document, they guarantee a unique context group for the site and a safe loading of trusted resources.

2.6.2. Preventing the measurement

Instead of fixing the data leakage at the source, academics have proposed to obfuscate the leakage by preventing attackers from detecting or measuring it. This can be either by removing access to measurement tools or randomizing execution to prevent monitoring.

2.6.2.1. Hardware Level

Many cache attacks exploit the deterministic nature of cache indexing, e.g., the usage of an eviction set in Prime+Probe. To prevent attacks, researchers have proposed randomized caches, obfuscating the leakage by randomizing the mapping from memory addresses to cache sets. Wang and Lee [WL07] proposed to use a dynamic random permutation table for each process, rendering other processes' cache access unpredictable. Liu and Lee [LL14] proposed random-fill caches, where cache misses fill the cache with randomized values in addresses next to the accessed address. However, both approaches require hardware-based lookup tables and are not scalable to large last-level caches. Computation-based approaches have been developed for large randomized caches. CEASER [Qur18] is a secured cache using

low-latency block ciphers as an indexation function. This address-set mapping is frequently refreshed by changing the cipher key. Trilla et al. [THAC18] propose another approach to cache randomization. Instead of randomizing the address-to-set mapping, they inject random timings to the cache behavior, lowering the risk of timing leakage. Purnal et al. [PGGV21] proposed a systematic approach to randomized caches, listing most major contributions.

2.6.2.2. System Level

The operating system can prevent data leakage by preventing the timing measurement. For cache attacks, if an attacker cannot distinguish cache hits from misses, the attack is not possible. Vattikonda et al. [VDS11] modify the native `rdtsc` instruction on virtualized systems. They reduce the timer by adding significant jitter, i.e., a random component to timing measurement. Similarly, StopWatch [LGR14] is a cloud-based architecture aiming at preventing co-residency timing side channels. Each VM is replicated, and I/O timings are computed on the median of all replicated VMs' timings.

2.6.2.3. Application Level

Applications can enforce randomization in their execution to prevent attackers from retrieving the full secrets. In particular, applications using sensitive secrets, e.g., keys in cryptographic computation, use blinding techniques to prevent timing information from being directly related to the key. Instead of using the key for operations such as scalar multiplication, the application modifies the key, generally with a xor operation, with a random factor. Then, after the computation, the result is returned to its original value. If the attacker retrieves information with a side-channel attack, they obtain information about the xorred key and not the actual key. This random component is computed at runtime and changes for each computation, preventing the attacker from reverting the changes.

2.6.2.4. Browser Level

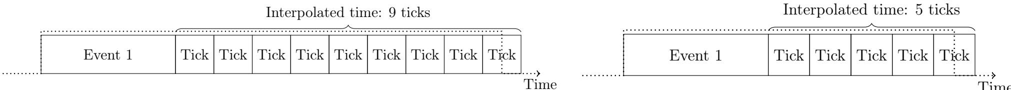
After the publication of microarchitectural side-channel attacks in the JavaScript sandbox [OKSK15], browser vendors have enforced timing-based countermeasures in client-side scripting languages. By removing access to high-resolution timers, browser vendors aim at reducing data leakage in generic JavaScript applications.

Mitigations on `performance.now()` Such changes have been applied to `performance.now()`, JavaScript high-resolution timer implementation. After the first JavaScript timing attack [OKSK15] in 2015, the W3C advised that the resolution should be reduced to 5 µs to mitigate timing attacks, and browsers followed [Ore15, K15].

However, in 2017 Schwarz et al. [SMGM17] demonstrated that it was still possible to recover high-resolution timers with the clamped resolution by using interpolation. This allowed an attacker to reimplement most timing attacks. In particular, they presented a clock interpolation-based timer with a resolution of 500 ns.

After the disclosure of Spectre [KHF+19], browser vendors went to greater lengths to mitigate timing attacks. Mozilla first clamped `performance.now()` to a resolution of 20 µs in Firefox 57.0.4 [Wag18] then furthermore to 2 ms in Firefox 59 [Bug18a].

Browser vendors then introduced jitter to `performance.now()`: Firefox 60 set the resolution to 1 ms and added a jitter of 1 ms range [Bug18b, Con20b], while Chrome 64 set the resolution to 100 µs with a jitter of 100 µs [Kyö18].



(a) The time between two clock edges is longer: the interpolated time is 9 ticks.

(b) The time between two clock edges is shorter: the interpolated time is 5 ticks.

Figure 2.15. – Impact of `performance.now()`'s jitter on clock interpolation: for the same event, the interpolated time changes between two measurements

The goal of this jitter is to prevent clock interpolation as presented by Schwarz et al. [SMGM17] and described in Section 2.3.3. Figure 2.15 presents the impact of jitter on interpolation. Clock interpolation does not measure the duration of an event *per se*, but rather the duration between the end of the event and the next clock edge. If the duration of a clock period is constant, this is equivalent to measuring the duration of the event. However, if the duration of a clock period is not constant, measuring twice the same event can yield different timings, rendering the measurement non-deterministic. If the jitter is higher than the needed resolution for timing attacks, i.e., sufficient to distinguish the two leaking timings, attackers cannot mount their attack.

However, such drastic countermeasures also impacted web development [Bug18c]. JavaScript-based animation or video games often require sub-millisecond timers, which were no longer available. These countermeasures were however temporary, until other countermeasures were developed. Indeed, because of the security added by site isolation, most browsers have re-allowed high-resolution timers under certain conditions.

Since Chrome 91, `performance.now()` has a resolution of 5 µs with a jitter of 5 µs under COOP/COEP [Sta], whereas, since version 79, Firefox has a resolution of 20 ± 20 µs without jitter, only when COOP/COEP is activated [Moz20b, Bug20].

The implementation of jitter differs across browsers. On Firefox 81, the jitter is computed using the SHA-256 hash function as follows [Moz20b]: the high-resolution timestamp is clamped to the lowest millisecond. The browser then uses SHA-256 on a tuple composed of the clamped time, a context-dependent seed, and a secret seed. It returns a random midpoint between the clamped time and the next millisecond. Depending on whether the precise timestamp is under or above this midpoint, the returned timestamp will be the lowest or highest millisecond, uniformly distributing clock edges between 0 and 2 ms. On Chromium, the jittered value is computed similarly, using murmur3 as the hash function [Good].

Mitigations on `SharedArrayBuffer` Due to the powerful threat this `SharedArrayBuffer`-based clock create, `SharedArrayBuffer` were disabled by default after the publication of Spectre [KHF+19] in Firefox 57.0.4 and all Chrome versions from 60. Once again, this measure is very restrictive for web developers and was only meant to be temporary. With the introduction of COOP/COEP and site isolation, browsers have re-enabled `SharedArrayBuffer`, claiming that access to a high-resolution timer is not a major threat in a strict isolation context. Both Chrome and Firefox current versions support `SharedArrayBuffer` only when COOP/COEP is

enabled [Gooc, Conf].

Other approaches Due to the complexity of completely removing timers in the browser, researchers have proposed more systematic solutions. One approach is to try and make the browser deterministic as prototyped by Cao et al. with DeterFox [CCLW17]. By transforming a physical clock into a logical one, they change the behavior of known timers so that they do not return the time a request takes but return the number of operations that are being executed. However, a deterministic browser would require significant architectural changes and would represent a significative performance overhead. Some programs need to access the actual physical clock of the system to function properly and having a deterministic clock would present a lot of hurdles for developers. Schwarz et al. [SLG18a] propose a fine-grained permission system to prevent JavaScript side-channel attacks. It allow to control the fuzziness of timers, as well as other software-based mitigations such as array index randomization, slower `SharedArrayBuffer` or disabling them, or removing JavaScript multithreading.

2.6.3. Detecting the attacks

Additionally to preventing the attack at its source, researchers have proposed methods to detect – and stop attacks at runtime.

2.6.3.1. Hardware and System Level

Initially a debugging and optimization tool, Hardware Performance Counters can be used to detect side-channel attacks during execution. While performance counters are purely microarchitectural features, the detection is generally handled by system handles on HPCs. Foreman [For18] classify performance-counter-based detection in three categories.

Signature-based detection focuses on detecting well-known threats in an unadaptative scenario. Chiappetta et al. [CSY16] proposed to detect Flush+Reload attacks based on the L3 cache usage. In a standard execution, a process should benefit from the cache with cache hits. However, when they fall victim to Flush+Reload-based attacks, due to the constant flushes, processes emit significantly more L3 cache misses than usual. Chiappetta et al. use this difference to detect attacks in real-time by comparing the evolution of L3 misses with performance counters between a signature of a standard, unattacked execution and the signature of the runtime execution. If the current execution presents more cache misses, it means that the process is under attack. In their paper presenting Flush+Flush [GMWM16], Gruss et al. show how these performance-counters-based detection methods are ineffective against Flush+Flush, as this attack primitive does not cause cache misses. This signature-based approach has the advantage of being simple to implement as only standard signatures are needed, but is not highly adaptative to new attacks or attacks on new implementations.

Heuristic-based approaches try to provide a more generalistic approach. They monitor malicious behaviors by setting thresholds on specific performance counters. When one or several thresholds are crossed, they detect possible attack threats. CacheShield is a heuristic-based approach, monitoring the quantity of cache misses with performance counters. When a threshold is crossed, the tool reports it as a possible attack. Aweke et al. [AYQ+16] proposed a HPC-based protection leveraging last-level cache latest misses, Load Latency, and Precise Store counters to detect when a DRAM row is being hammered. When this detection occurs, the adjacent rows are refreshed. HexPADS [Pay16a] is an automated framework to detect

outstanding behavior and classify them as potential attacks. It is able to detect RowHammer attacks or cache side channels. Heuristic-based approaches allow for more genericity in the detection, but may yield a higher rate of false positives than other approaches as standard execution may accidentally cross thresholds.

Machine-learning-based detection is emerging as a compromise between the two previous approaches. It has been used by Chiappetta et al. [CSY16] to enhance detection using their signature-based approach. Prada et al. [PIO19] trains a model using L3 cache misses performance counters to detect Flush+Reload side channels against AES. By monitoring last-level cache events as well as branch mispredictions, Li and Gaudiot [LG18] propose a machine-learning-based approach to detect Spectre-PHT type attacks. Wang et al. [WSS+20] systematize the approach of machine learning on HPC-based detection to compare the efficiency, accuracy, and performance overhead of various sampling methods, training datasets, or models.

2.7. Browser Fingerprinting

A *browser fingerprint* is a collection of information allowing to identify, uniquely or not, an instance of a browser. Browser fingerprinting is closely related to side channels: an agent extracts legit information, e.g., the browser version or language for browser fingerprinting, to distort them into another usage. While side channels often aim at stealing secrets or monitoring a user, browser fingerprinting aim at circumventing the web's anonymity. This section provides a brief introduction to browser fingerprinting, necessary to comprehend the privacy implications of Chapter 5.

2.7.1. Usages of Browser Fingerprinting

The browser's profile created by browser fingerprinting allows agents to break the anonymity of the web. This fingerprint can be used for different purposes, legitimate or malicious:

Web tracking: Tracking is probably the most widespread fingerprint application. An agent can create a fingerprint for a user and recognize it on various websites, thus monitoring its activity. It allows, for instance, to provide targeted advertisement. This tracking fingerprinting is particularly interesting on the modern web, where other tracking methods, e.g., cookies, are regulated by laws.

Authentication: For more legitimate purposes, fingerprints can be used as a method to enhance authentication. In addition to a classic authentication, e.g., with a password, a website can retrieve the user's fingerprint and compare it with previous authentications. If the fingerprint changed, the website can ask for a more secure mean of authentication.

Bot identification: Users' fingerprints result from a set of parameters emerging from a normal utilization of a computer and browser. A computer-controlled browser can lack a coherent human-like fingerprint, and sites can use this difference to detect bots and reduce risks of denial of service attacks or web crawling.

Vulnerability discovery: Browser fingerprinting can reveal information about the system that is supposed to be hidden, e.g., the version number of software or OS. A malicious user can use this information to find specific exploits against that system, thus reinforcing the threat posed by attackers.

2.7.2. Attributes of Fingerprints

A fingerprint is often a set of various attributes ranging from browser values to hardware information. A fingerprint, or more precisely an attribute, can be more interesting than others with certain properties:

Uniqueness: The end goal of fingerprinting is uniquely identifying a user. To that extent, a unique fingerprint is necessary. Unique attributes are rare, and this uniqueness is generally obtained by collecting multiple attributes. However, an attribute can provide more uniqueness if it allows to reduce the scope of users on its own.

Stability: Any change in an attribute value changes the fingerprint and therefore breaks user identification. However, relying on software fingerprinting means that attributes are constantly changing (e.g., the browser version in the User Agent). Vastel et al. [VLRR18] showed that it is possible to link two fingerprints that are slightly different from each other through heuristics. Therefore, uniqueness is less critical than stability to link fingerprints for a single attribute.

2.7.3. Software Fingerprinting Techniques

Software attributes are by far the most studied in the literature. In 2009, Mayer [May09] demonstrated that information sent by the browser to the server for configuration purposes, e.g., screen size, could lead to the deanonymization of users. This finding was extended in 2010 [Eck10] in a larger-scale approach, considering HTTP headers, JavaScript active fingerprinting, and plugins such as Flash. This study showed that 95% of collected fingerprints were unique.

Since the publication of these original papers, the variety of software-based fingerprint attributes only grew. Fingerprints used various features such as the Canvas API [Sto13, AEE+14], fonts [FE15], battery status information [OACD15], WebGL [Sto13] or installed browser extensions [SVS17, SSB17b, SLKN19].

2.7.4. Hardware Fingerprinting Techniques

Instead of exploiting software-based information, an attacker can leverage side channels to infer hardware-related attributes. This fingerprinting method is typically active, i.e., the attributes are not directly given by the browser but rather inferred from computation, e.g., a JavaScript code. Nakibly et al. [NSY15] proposed various ideas to identify a user based on fingerprints of their audio system or GPU. Sanchez-Rola et al. [SSB18] leveraged imperfections in the crystal of processor clocks to identify users. More recently, Laor et al. [LMD+22] showed that the difference in execution time between different execution units in individual GPUs can serve as a unique and robust fingerprint.

As they are not static, hardware fingerprints can be harder to obtain than software fingerprints but are more stable as users rarely change hardware components and are less likely to be spoofed as a user cannot modify them as easily as an HTTP header.

3 High Resolution Timers in the Browser

JavaScript-based timing attacks have been greatly explored over the last few years. They rely on subtle timing differences to infer information that should not be available inside of the JavaScript sandbox. In reaction to these attacks, the W3C and browser vendors have implemented several countermeasures, with an important focus on JavaScript timers. In 2015, Oren et al. [OKSK15] implemented a fully JavaScript-based cache attack, running entirely in the browser. Based on Prime+Probe [LYG+15], it allows an attacker to track user behavior and recover information belonging to other processes running on the same system. This contribution opened the scope for more critical attacks, exploiting hardware components from the restricted JavaScript sandbox. To try and mitigate JavaScript-based timing attacks, browser vendors have developed countermeasures specifically targeting timers. Notably, they decreased timers' resolution to make them less precise and introduced jitter to add noise to measurements. Other security features like site isolation [RMO19] were added to reinforce the browser's security and act as a novel line of defense against timing attacks. Amid all these changes, it can be hard to keep track of all the different evolutions that browsers underwent: each browser vendor implements different countermeasures, and decisions can be taken in a hurry after the publication of critical vulnerabilities. For instance, in 2018, drastic timer-based countermeasures were taken only a few days after the publication of Spectre [KHF+19]. These patches can be hard to track, and the quantitative aspect of such countermeasures has not been appropriately studied.

In this chapter, we aim to provide a clear view of the vulnerability of browsers to timing attacks. In the first part, we take a broad look at the research done in the area to systematize it. We provide a taxonomy of attacks with their prerequisites and classify the countermeasures based on their target resources. In the second part, our goal is to assess the actual efficiency of timing-based countermeasures after seeing how much they changed over the years. In order to gain proper insight, we implemented our own `performance.rdtsc()` high-precision timer into Chromium and Firefox so that we can deconstruct studied timers into their most basic blocks. With its help, we identified that recent countermeasures present significant advances against timing attacks, but they also present noticeable steps back. For example, with the introduction of COOP/COEP, Firefox 79 gained a robust resource isolation mechanism but lost a lot with regard to timing attacks. Before, an attacker needed several minutes to build an eviction set to conduct an attack. Now, with the resolution changed from 1 ms to 20 µs, we show that it can be set up in just a single second. Another problem is the recent reintroduction of `SharedArrayBuffer` after it was deactivated due to the disclosure of Spectre [KHF+19]. Its presence introduces a real security risk because a malicious script can abuse it by creating a very powerful timer that has an incredibly high resolution with very low overhead. By lowering timing-based countermeasures, all the prerequisites for large classes of timing attacks are met, meaning that these attacks can theoretically be implemented.

Contributions This chapter makes the following contributions:

- We provide a classification of prerequisites for timing attacks and present the most notable classes of timing attacks (Section 3.1).
- We classify countermeasures based on the resources they target (Section 3.2).
- We present tools to analyze timers and the threat they pose for timing attacks. Notably, we detail our custom timers and automatic tests that measure the resolution and measurement overhead of several built-in timers, which can easily be reproduced for other systems and future browser versions (Section 3.3).
- We present a longitudinal study of browsers' timing-based countermeasures. (Section 3.4).

Update to the results of this contribution This contribution, including the results and the conclusion and discussion, stem from an article written in 2020. Due to the rapid development of browsers and research in security, several results may have changed since. Although these changes do not drastically impact the conclusions of this article, Section 3.7 presents the latest changes and discusses their consequences.

3.1. Timing attacks in browsers

Timing attacks in browsers are a large class of attacks exploiting timing differences in computations in order to infer private information. As they are mainly based on JavaScript, the attacker code will, by design, always be executed on the victim's hardware. In this section, we aim at systematizing timing attacks in browsers, by classifying prerequisites for different classes of attacks.

3.1.1. Attack prerequisites

We systematize the major timing attacks prerequisites in order to classify them and better understand the outline of attacks. We have identified the following prerequisites:

P1 High-resolution timers,

P2 Shared hardware resources,

P3 Transient execution,

P4 Shared system resources,

P5 Shared browser features.

P1: High-resolution timers To mount timing attacks, an attacker must be able to distinguish between data-leaking events with their timings. To do so, they need access to high-resolution timers. JavaScript offer several timers presented in Section 2.3, but this chapter focus on the two timers with the best resolution: `performance.now()` interpolation and `SharedArrayBuffer`.

Table 3.1. – Comparison of attacks prerequisites. All attacks require P1, but with different ranges of resolutions.

	Required resolution (P1)	P2	P3	P4	P5
Spy in the sandbox [OKSK15]	100 ns	●	○	○	○
Rowhammer.js [GMM16]	100 ns	●	○	○	○
Website fingerprinting [SKH+19]	10 – 100 ms	●	○	○	○
DRAM covert channel [SMGM17]	10 ns	●	○	○	○
Breaking ASLR [GRB+17]	100 ns	●	○	○	○
Spectre [KHF+19]	100 ns	●	●	●	○
ret2spec [MR18][1]	100 ns	●	●	●	○
RIDL [vSMÖ+19]	100 ns	●	●	●	○
Store-to-Leak [CGG+19]	100 ns	●	●	●	○
Memory deduplication [GBM15]	1 µs	○	○	●	○
Loophole [VK17][2]	100 µs	○	○	○	●
Extension side channel [vGJ17]	100 ns	○	○	○	●

P2: Shared hardware resources Modern CPUs present major microarchitectural optimizations in order to compute rapidly and efficiently, often solely created with performance in mind. Since hardware is situated below software security rings and components are often shared between processes, contention at the microarchitectural level can be the root cause of software-based side-channel attacks. By design, these attacks are also less fixable than software.

P3: Transient execution Transient instructions are instructions that are executed but never committed to the architecture. They can be produced by several factors, such as speculative execution or faulting memory accesses.

P4: Shared system resources In a browser, system resources are often shared between contexts. Memory in particular can be shared between all code running in the same browser process.

P5: Shared browser features A lot of purely software features are shared between contexts or tabs in browsers. This is the case for history, cookies, event loops, or renderers, for example. The difference in timings of such features' operation can leak information between tabs and inform about a user's behavior.

3.1.2. Attack classes

We now give an overview of the major classes of timing attacks in browsers, along with their major characteristics and prerequisites. Table 3.1 illustrates the prerequisites for a sample of state-of-the-art timing attacks in browsers.

Hardware-based side-channel attacks Hardware-based side-channel attacks exploit hardware components shared by all processes in the system. By measuring the computation time of specific operations, an attacker can infer the state of certain components. These attacks share two major prerequisites:

P1 High-resolution timers,

P2 Shared hardware components, e.g., cache, DRAM.

The resolution of required timers varies in function of the attacked hardware, but typically spans from $10 - 100\,\text{ns}$ in order to potentially distinguish different states in the hardware.

Transient execution attacks With Spectre [KHF+19], Kocher et al. paved the way for a new class of attacks, exploiting transient execution to leak protected data. Some of these attacks can be implemented in browsers in JavaScript, including Spectre-PHT [CBS+19]. This class of attacks is very wide, but shares some prerequisites:

P1+P2 A hardware covert channel to extract leaked data,

P3 Transient execution,

P4 Shared system resources—finding secrets to leak.

P3 is a wide prerequisite: the Spectre attack alone possesses many variants, each targeting different optimizations [CBS+19]. Most crucially, different attacks target different secrets to leak (P4), i.e., in the same address space, or across address spaces.

Attacks based on system resources By exploiting shared system resources, an attacker can also retrieve information using a side-channel attack. These attacks share the following prerequisites:

P1 High-resolution timers,

P4 Shared system resources.

Attacks based on browser resources In order to have a uniform and efficient browsing experience, most browsers share information between tabs or processes. This includes, among others, browsing history, browser extensions, or event loops. However, this sharing of information, even if not directly reachable in JavaScript, can leak private information to a malicious site. This class of attacks shares two prerequisites:

P1 High-resolution timers,

P5 Shared browser resources.

[1] ret2spec can still break ASLR with site isolation.

[2] Since each process has its own event loop, the attack is not implementable on recent Chrome versions.

3.2. Countermeasures in browsers

In this section, we systematize the different countermeasures proposed by academics or browser vendors. We have categorized such countermeasures into three classes:

C1 Isolation-based countermeasures,

C2 Timing-based countermeasures,

C3 Browser resources based.

3.2.1. C1: Isolation

Isolation has been established as a staple of browser security with measures such as site isolation or COOP/COEP, presented in Section 2.6.1.4.

While these countermeasures were mostly implemented to prevent transient execution attacks, especially Spectre-PHT [KHF+19], they do not prevent other timing attacks, e.g., microarchitectural side channels or attacks exploiting browser features as, by design, they are not meant to mitigate P1, P2, or P3. As site isolation isolates the process' attack space, they also do not mitigate P4 for transient execution attacks targeting cross-address-space data, such as RIDL [vSMÖ+19].

3.2.2. C2: Timers

The common prerequisite for timing attacks is access to timers. By removing timers, or lowering their resolution, most timing attacks would theoretically be mitigated. While the needed resolution depends on the attack, most hardware and transient attacks require high-resolution timers (P1), around 10 to 100 ns. Removing such high-resolution timers would theoretically mitigate these attacks.

However, even by reducing timers' resolution, interpolation is still possible as long as attackers have access to timers with a constant time free-running operation. To mitigate clock interpolation in browsers, adding a random jitter to API timers was proposed [SMGM17, KS16]. Adding jitter to a measurement is equivalent to having clock periods of different times (with an average around the real clock period). This means that the interpolated time would vary significantly between clock periods for the same event, hence reducing the precision of clock interpolation.

Another widely-adopted countermeasure was to disable `SharedArrayBuffer` to prevent attackers from using them as high-resolution clocks.

Firefox 81's `performance.now()` has a resolution of 20 µs without jitter, only when COOP/COEP is activated [Moz20b, Bug20], and a resolution of 1 ms when COOP/COEP is not activated. It offers access to `SharedArrayBuffer` only when COOP/COEP is enabled. Since Chrome 72, `SharedArrayBuffer` is enabled and `performance.now()` has a resolution of 5 µs with a jitter of 5 µs under site isolation (which is present by default)[3].

[3] Since the original publication of this contribution, newer Chrome versions changed the policy about timer security, and now these changes require COOP/COEP to be activated to recover `SharedArrayBuffer` and high-resolution timers. See Section 3.7 for a short update on later changes.

3.2.3. C3: Browser resources

When some timing leakages in the browser are not fixed by a more general approach like C1, there is a need to issue patches that specifically target them. For example, the history sniffing attack detailed by Paul Stone in 2013 [Sto13] was only fixed in Firefox in 2020 by issuing repaints on both visited and unvisited elements [Moz20a, Moz19]. Regarding extension fingerprinting, the timing attacks detailed by Sanchez-Rola et al. [SSB17c] and Van Goethem and Joosen [vGJ17] were fixed by the Chromium team by changing how the checks for web-accessible resources were made to prevent early-out exits [Chr17a, Chr17b].

3.2.4. State of browser countermeasures

Figure 3.1 illustrates the evolution of browsers' main countermeasures and significant state-of-the-art attacks. P2 and P3 are not mitigated in browsers. Site isolation and COOP/COEP are important security updates on browsers, but do not mitigate hardware side-channel attacks and transient execution attacks other than Spectre-PHT. Furthermore, V8 developers claim that software fixes for transient execution attacks are "an unsustainable path" [TS] as fixes on P3 are too performance consuming, and fixes on P4 only apply to specific attacks. Mitigation for P3 must be implemented at least at the OS level, if not at the hardware level.

The common prerequisite of timing attacks is P1, i.e., access to high-resolution timers. This means that C2, timing-based countermeasures, are the only generic defense for timing attacks, including future timing attacks.

Changes in timer-based countermeasures were motivated by a compromise between security—less effective timers mean less threatening attacks—and usability. Indeed, countermeasures had major implications on web development and were meant to be temporary. `SharedArrayBuffer` are a powerful tool to build more complex websites, and access to high-resolution timers is necessary for some fields of web design, such as animation or monitoring performances. There is a clear trade-off between strengthening the timers for security, and weakening them for easier development. However, we have found no quantitative study of the impact of the changes in values, especially regarding API timers' resolution and jitter. Finding when and why each countermeasure was deployed often requires a deep dive into browser bug trackers.

With all the changes brought to countermeasures, especially timing-based, it is not clear to what extent P1 is mitigated, i.e., to what extent browsers are vulnerable to most timing attacks, and whether an optimal value for resolution and jitter exists. In the following sections, we evaluate the efficiency of timing-based countermeasures.

3.3. Evaluation tools

In this section, we present our threat model, and the properties of timers we are interested in: their resolution and measurement overhead. We focus on three timers or variants: `performance.now()` interpolated, `performance.now()` interpolated and amplified, and `SharedArrayBuffer`.

3.3.1. Threat model

We evaluate two popular browsers that are Mozilla Firefox and Google Chrome, letting aside Safari, Edge, or Tor Browser. It should be noted that we study the desktop version of these browsers, as all timing attacks we look at were not performed on mobile devices.

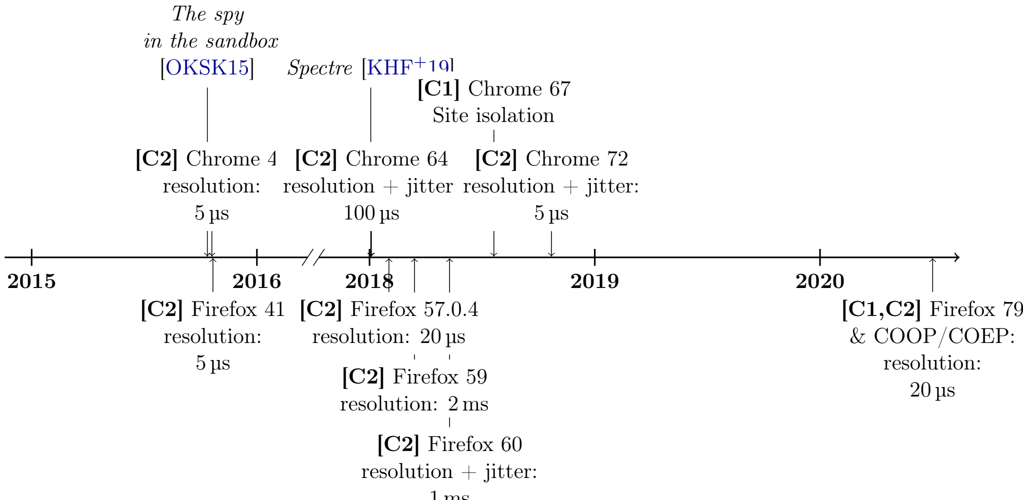


Figure 3.1. – Timeline of timing attacks and browser countermeasures. Items in italics are significant attacks that caused the changes. Items preceded by **[C1]** are isolation-based countermeasures, and **[C2]** are timing-based countermeasures

JavaScript-based timing attacks can be used in several threat models, each offering a different range of possibilities. First-party attacks, where a user visits a malicious website, offer the most possibilities to the attacker. As the attacker can setup COOP/COEP as she wishes, she can freely use the unrestricted timers, based either on `performance.now()` or `SharedArrayBuffer`. In this model with Firefox 81, an attacker has access to `performance.now()` with 20 µs resolution and no jitter and to `SharedArrayBuffer`. However, the attacker has to redirect users to her malicious website.

With third-party attacks, a user visits a legitimate website that has, e.g., a malicious advertisement controlled by the attacker. This allows the JavaScript code of the ad to be run on the user's machine. As opposed to the first-party model, the attacker does not control the top-level page. The attacker therefore cannot set up COOP/COEP. Without these flags, an attacker on Firefox 81 only has access to `performance.now()` clamped and jittered at 1 ms and no access to `SharedArrayBuffer`. This model presents a major threat surface, as it can be implemented on massively visited websites, e.g., Twitter or Facebook.

On Chrome 84, timer-based countermeasures are independent of COOP/COEP. This means that the first and third-party models offer the same timing possibilities, i.e., access to `SharedArrayBuffer` and `performance.now()` with a 5 µs resolution and jitter. However, Chrome 84 implements site isolation, which prevents certain transient execution attacks.

3.3.2. Measurement tools

One challenge is that measuring properties of high-resolution timers requires timers with an even higher resolution and precision, which are not available in standard browser releases. To solve this issue, we built a custom version of Firefox and Chromium that adds a new JavaScript method we call `performance.rdtsc()`, which executes the `rdtsc` instruction. The `rdtsc` instruction reads a CPU timestamp counter and returns it. As `rdtsc` is a cycle-accurate timer, we can use it to evaluate real-world auxiliary timers. Specifically, we use it to measure

the resolution of both `performance.now()` interpolation and `SharedArrayBuffer`, as well as their measurement overhead.

We implemented our custom `rdtsc` builds in Firefox 81 and Chromium 84, built from sources on the same system. We disabled debug flags to get a version as close to the release version as possible. We modified files of the `performance` API [Con20a] to add a new public method, `performance.rdtsc()`, which executes the native `rdtsc` and returns its value. However, `rdtsc` can be reordered by the out-of-order execution, therefore two calls to `performance.rdtsc()` would often be executed together, hence not timing an event. For instance, two calls to `performance.rdtsc()` with a call to `performance.now()` between would return the same timing difference as two subsequent calls to `performance.rdtsc()`. We fixed this by adding memory fences (`mfence` instructions) in our `performance.rdtsc()` method, before and after the call to `rdtsc`. Memory fences force the CPU to compute all operations preceding the fence before executing operations succeeding the fence. This prevents out-of-order execution of parts of code where the order of execution is critical. Appendix A.1 illustrates our implementation of `performance.rdtsc()` for Firefox 81.

The `mfence` instructions add a constant overhead to the returned timestamp. Overhead is also added by the browser handling of the native code. On our system, the total overhead is around 1000 cycles. In all our following measurements, we measured the cycle difference between two events, and then the cycle difference between two subsequent `performance.rdtsc()`. This allows us to estimate the overhead for the current state of the CPU, depending on noise or core frequency. By removing this overhead, we retrieve a more reliable estimation of the event time in cycles.

3.3.3. Resolution

The resolution of a timer is the smallest value that it can measure. The smaller the resolution, the more precise the timer is. However, precisely evaluating the resolution of a JavaScript timer is a challenge as this requires an already precise timer.

`performance.now()` interpolation To evaluate the resolution of `performance.now()`, we evaluate the maximum number of times we can increment a variable between two clock edges. By dividing the clock edge duration by this number of ticks, or measuring the time it takes to increment, we learn the shortest event that we can measure. However, because of the jitter, this value alone is not representative. It can vary significantly between measurements. Hence the importance of the standard deviation of the resolution.

It is also important to note that the duration of a "tick", an increment, can vary from one browser version to another. The main factor of this varying duration is due to the implementation of clock interpolation: after every increment, we check if `performance.now()` timestamp has changed. As the computation of the pseudo-random jitter often uses a hash function, it significantly increases the `performance.now()` computation time. Logically, the number of possible increments during the same duration changes according to this computation time.

`performance.now()` amplification As the resolution is downgraded by pseudo-random values, we also study the impact of amplification. By repeating the measurement, an attacker can average the results and reduce the impact of the jitter. This amplification allows an attacker to recover a higher resolution. As each repetition can increase the resolution, granting an

absolute resolution for an amplification clock is illogical. The attacker has to compromise between the resolution and the number of repetitions she can afford.

In the following sections, we call error rate for cache hit/miss discrimination, the ratio of false hits, i.e., misses computed as hits, and false misses, i.e., hits computed as misses over the total number of experiments. This error rate allows to evaluate the efficiency of a timer in a real-world cache timing attack. We assume that a timer with a 5% error rate has a resolution sufficient to implement cache timing attacks, hence a resolution of around $10-100$ ns. This rate will be used as a standard for amplification in the following sections. We compute this rate by causing ourselves cache hits by calling a variable repeatedly and cache misses by calling a variable after evicting it from the cache by browsing through a large array.

Amplifying the results by repeating measurements is, however, not adapted to all attacks. Specifically, it only works when the attacker controls the event that creates the timing difference. For instance, in the case of a covert channel, the attacker can recreate the conditions for the same measurement several times. Some monitoring attacks do not tolerate repetitions, as the attacker cannot recreate the same conditions for the measurement. For instance, RIDL [vSMÖ+19] steals in-flight data, and cannot select which data is in flight.

`SharedArrayBuffer` For `SharedArrayBuffer`, the resolution is the time of a shared increment. The faster the increment, the higher the resolution. Other factors can impact the resolution of the timer, such as the computation time of reading a value from the array or potential concurrent accesses. Multithreading is handled differently in different browsers and this can be a potential source of randomness on the timestamp.

3.3.4. Measurement overhead

While the resolution indicates how precise a timer can be, an overhead is always incurred during a time measurement. In our study, we consider the following ones:

- Setting up a timer and handling it (starting it, stopping it, retrieving the results) always adds an overhead. For `performance.now()`, it is very low as we rely on a built-in API in the browser. For `SharedArrayBuffer`, it is also comparatively low as starting a worker is very fast and communication with it is immediate.
- The repetition of ticks in timer interpolation (see Figure 2.8) is an overhead. Indeed, even if we measure the time of a short event, we still need to wait for the next clock edge of the timer that we rely on to retrieve the results. This is why, for `performance.now()` interpolation, the resolution can be very low but the overhead can be large as we are bound by `performance.now()` resolution.

To evaluate the measurement overhead of a given method, we use our custom `performance.rdtsc()` to measure the time difference between the start and the end of the measurement, without any event in between. A high measurement overhead does not necessarily prevent the attack but it plays a major role in the attack severity.

Ideal bit rate A covert channel created with a high measurement overhead will yield a lower bit rate than one with a low measurement overhead, i.e., the time to receive one bit will be higher with a high measurement overhead. Assuming each measure of time corresponds to a bit of information, and that the time the operation takes t_{op} is insignificant with respect to the overhead t_{oh}, we can define the ideal bit rate of such a timer to $\frac{1}{t_{oh}}$ bit/s.

Table 3.2. – Comparison of timers' resolution, measurement overhead and ideal bit rate for an error rate of 5%. We used a frequency of 1.60 GHz and the resolution is displayed in ns. We can observe that the `SharedArrayBuffer`-based timer is by far the most efficient, as it offers a better resolution and a lower measurement overhead than timers based on `performance.now()` on all browsers. Timers based on `performance.now()` clocks are still highly effective in Chrome 84 and Firefox 81 with COOP/COEP.

Browser	Timer	Resolution [cycles]	Converted resolution	Measurement overhead [cycles]	Ideal bit rate [bit/s]
Chrome 84	`SharedArrayBuffer`	20	10 ns	40	1×10^8
	`performance.now()` interpolation [4]	100-1000	100 ns	7.2×10^4	22×10^4
Firefox 81	`SharedArrayBuffer`	20	10 ns	42	1×10^8
	`performance.now()` interpolation	100-1000	100 ns	2.9×10^7	60
	Interpolation with COOP/COEP	100-1000	100 ns	7×10^4	22×10^4

Eviction set Measurement time, and therefore measurement overhead, also impacts other attacks. For instance, building an eviction set in JavaScript [VKM19], has a complexity in time measurements of $\mathcal{O}(C)$ where C is the size of the cache. Standard L3 caches have a size of several megabytes. In practice, on our system, an attacker must measure latency around 100 000 times in order to build an eviction set. We use the computation of an eviction set as a standard as it is a critical step of attacks based on Prime+Probe. A high measurement time therefore leads to huge eviction set computation times, which may not be available to an attacker in a real-world scenario where a user only spends a few seconds or minutes on a web page.

3.4. Results

In this section, we present the results of our longitudinal studies of `performance.now()` and `SharedArrayBuffer` timers over many browser versions, as well as the impact of changes in countermeasures on state-of-the-art attacks. Table 3.2 presents the results of our comparative study.

3.4.1. Experimental setup

We ran measurements on a machine with an Intel CPU i5-8365U (Whiskey Lake generation) with 1.60 GHz frequency, under Fedora 31.

We performed experiments using every major release versions of Firefox from 53 (2017)

[4] COOP/COEP has no impact on timers in Chrome 84.

Table 3.3. – Duration of a tick, using `performance.rdtsc()`.

Browser	Average tick duration [cycles]	Standard Deviation [cycles]
Firefox 81	200	10
Unjittered Firefox 81	100	9
Chrome 84	150	9

to 80 (2020) and Chrome 48 (2016) to 84 (2020)[5]. We used Selenium WebDriver [sel20] and Python to automate tests. New versions of said browsers can easily be included in the test routine to see the evolution of timers without a deep dive in documentation and source code. Our scripts access test pages hosted on a local server, each page containing our JavaScript benchmark code. Our scripts also handle browser evolution, e.g., the different flags required by the use of `SharedArrayBuffer`, or the eventual activation of COOP/COEP.

3.4.2. Longitudinal study of `performance.now()` interpolation

3.4.2.1. Simple interpolation

Resolution We seek to measure the resolution of the `performance.now()` method when interpolated. Two factors influence this resolution: (1) the duration of a tick, and (2) the jitter.

The shorter the duration of a tick, the higher the resolution. Table 3.3 illustrates the duration of a tick on different browsers and versions. It is important to note that while the increment part of the tick has a relatively steady computation time through browser versions, the computation time of the call to `performance.now()` varies a lot depending on the version. For instance, on Firefox 81, a call lasts around 200 `performance.rdtsc()` cycles, while it only takes 100 cycles on an unjittered version. At this scale, the jitter is indeed a time-consuming operation as the browser uses a hash function to compute a random midpoint.

To understand the impact of resolution and jitter on measurements, we measure the number of ticks between two clock edges. This is equivalent to measuring no event with a `performance.now()` interpolation-based clock. Without any jitter, the number of ticks is always the same. Since the standard deviation is low, an accurate timing can be retrieved [SMGM17]. With jitter, the story is obviously different. If the resolution of `performance.now()` is low, the time between two clock edges increases, so the number of ticks increases. If the jitter is high, the number of ticks will be spread on a wide range of values between measurements, and therefore the standard deviation increases. Table 3.4 illustrates this evolution by showing statistics about the average number of ticks between two clock edges. As a reminder, a tick is composed of an increment as well as a call to `performance.now()`. The results were computed using 100 000 samples. We see that jittered versions often have a high variance to average ratio, e.g., on Firefox 81 without COOP/COEP, the standard deviation is half of the average.

[5] Some releases of Chrome (versions 65 to 6) were unavailable on our system. We replaced them with the equivalent Chromium version. To our knowledge, the timer implementations are the same on both browsers.
[6] Without COOP/COEP for Firefox 81 and later.
[7] Versions 58 and 59 experienced many timer changes and do not appear here.

Table 3.4. – Comparison of the average number of ticks per browser. The duration of a tick can vary according to the browser or the presence of jitter.

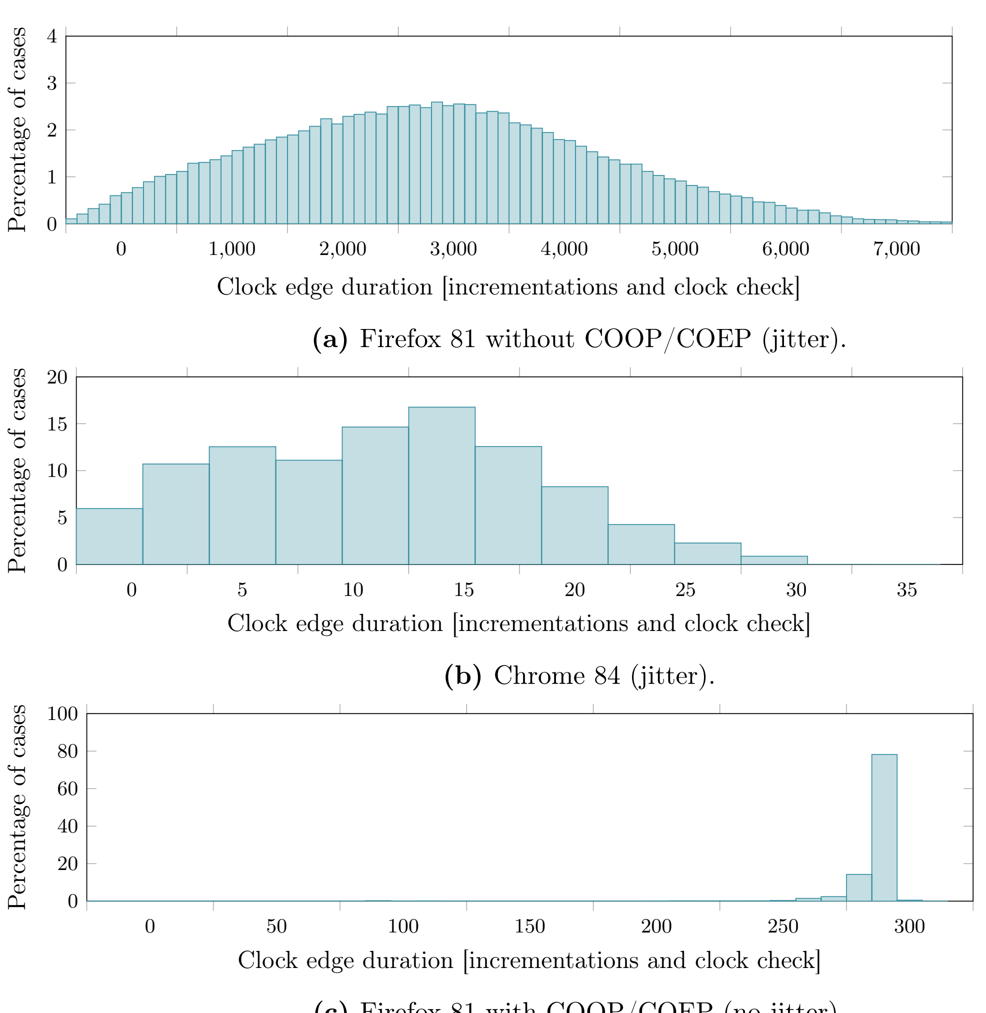
Browser	Average number of ticks	Standard Deviation	Announced resolution	Jitter
Firefox 81-latest + COOP/COEP	300	40	20 µs	○
Firefox 60-latest[6]	3310	1510	1 ms	●
Firefox 41-57[7]	70	10	5 µs	○
Chrome 72 and later	12	7	5 µs	●
Chrome 64-71	400	200	100 µs	●
Chrome 64 and former	10	2.5	20 µs	○

Figure 3.2a and Figure 3.2b illustrate the variation of the number of ticks in a single clock period for Firefox 81 and Chrome 84 respectively. The data follows a triangle distribution, with the top value at around 3300 and 12 ticks respectively. A precise timestamp can be clamped to a certain value if the timestamp is smaller than the clamped value but higher than the random midpoint, or if the timestamp is higher than the clamped value but lower than the random midpoint. As both midpoints are computed following a uniform distribution, the result of the sum of these two distributions is a triangle distribution. On the contrary, as can be seen on Figure 3.2c, the behavior of an unjittered Firefox is very different. The distribution of ticks in a clock period is grouped between 280 and 320 and does not follow a triangle distribution. The difference in the number of ticks stems from the different base resolutions and jitters. Knowing the distribution of the number of ticks in a single clock period means that an attacker can gather more information from a single measurement.

The resolution for the different versions varies drastically between browsers. Figure 3.3 illustrates the timings for cache misses and cache hits by using `performance.now()` interpolation on Firefox 81 with COOP/COEP and Chrome 84. Distributions must be differentiated in order to implement cache attacks. Note that, contrary to what we expect, hit timings are higher to miss timings on this graph. This is because `performance.now()` interpolation measures the time between the end of the event and the end of the clock edge. This means that a long event will yield a short interpolated time, whereas a quick event will yield a long interpolated time. The lack of jitter on Firefox has an impact on the histogram, where the difference between cache hits and cache misses is clearer.

On Firefox 81 without COOP/COEP, an attacker has a 30% error rate on distinguishing cache hits from cache misses when targeting a sub-100 ns resolution. On Chrome 84, regardless of COOP/COEP, the error rate stands at 20%. This makes for a highly unreliable clock, and drastically hinders the development of attacks. On Firefox 81 with COOP/COEP, the lack of jitter drops this error rate to only 11%.

The lower the error rate, the more threatening timing attacks with this timer are. This means that, with interpolation alone, timers based on `performance.now()` on Firefox 81 with COOP/COEP offer the best timer in terms of resolution, as they allow to implement cache-based timing attacks with a relatively low error rate. Error rates of Chrome 84 and Firefox 81 without COOP/COEP are too error-prone to implement attacks that require high precision.



(a) Firefox 81 without COOP/COEP (jitter).

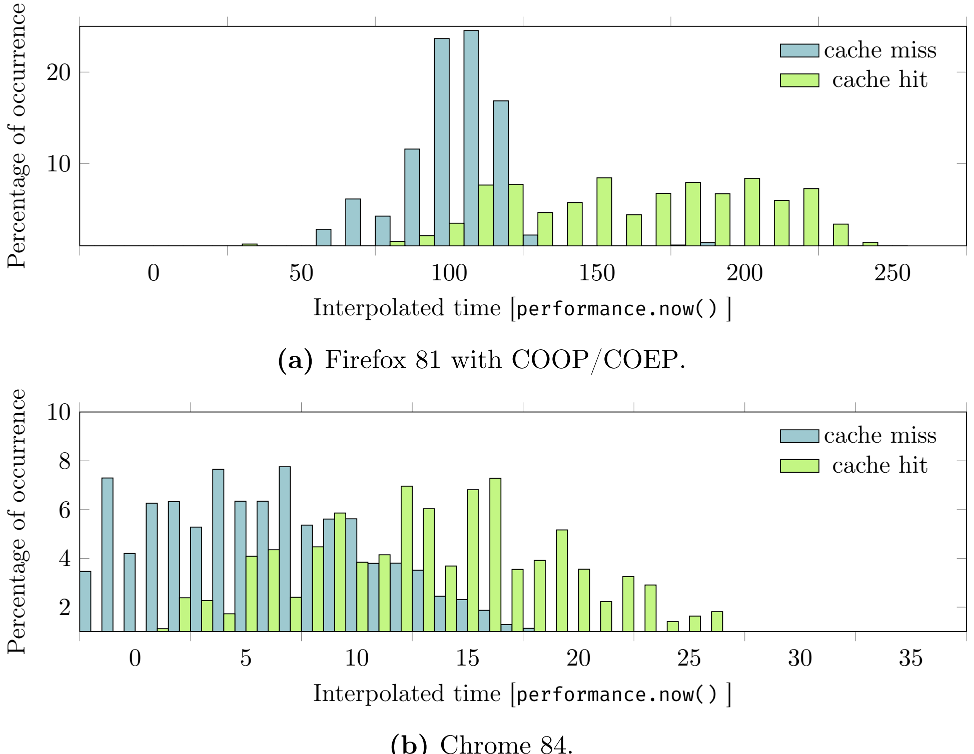
(b) Chrome 84 (jitter).

(c) Firefox 81 with COOP/COEP (no jitter).

Figure 3.2. – Distribution of the number of ticks in a single clock edge.

Measurement overhead We now evaluate the measurement overhead on Firefox 81 and Chrome 84. We use our custom `performance.rdtsc()` to retrieve the timestamp before and after the measurement of an empty event with `performance.now()` interpolated. We found that, on Firefox 81 without COOP/COEP, the measurement overhead averages around 1.8 million cycles. On our 1.60 GHz CPU, this represents around 1.1 ms. This is coherent with the announced resolution, 1 ms, as the measurement always takes at least this time between two clock edges. On Chrome 84, a measurement using interpolation takes 9,000 cycles, corresponding to 5.5 µs with our average frequency. On Firefox 81 with COOP/COEP, a measurement using interpolation takes 35,000 cycles, averaging to 21 µs on our system. The slight difference may come from the potential change in CPU frequency under a lot of calculations and noise.

With interpolation alone, Chrome 84 has the shorter measurement overhead, which allows implementing attacks that run faster. Namely, a cache covert channel built with this timer could ideally retrieve a resolution of 180 kbit/s, against 50 kbit/s for Firefox with COOP/COEP and 900 bit/s for Firefox without COOP/COEP. However, the ideal bit rate does not



(a) Firefox 81 with COOP/COEP.

(b) Chrome 84.

Figure 3.3. – Histogram for cache hits and cache misses.

take into account the different error rates between browsers.

Conclusion Although Chrome 84 offers the best base resolution with `performance.now()` interpolation, the jitter causes a high error rate for high-resolution attacks. This error rate would slow, if not prevent, the implementation of attacks using interpolation alone. Due to the lack of jitter, Firefox 81, with COOP/COEP enabled, offers the lowest error rate with interpolation. It could be used to build error-tolerant cache attacks, such as a covert channel. Firefox 81 without COOP/COEP has a high error rate (30%) and a high measurement time, rendering it inefficient for timing attacks.

With interpolation alone, Firefox 81 with COOP/COEP is the only browser where an attacker can build clocks based on `performance.now()` that can efficiently run timing attacks.

3.4.2.2. Interpolation and amplification

The jitter being longer than most of the events we wish to time, it is not possible to simply use clock interpolation to retrieve a high resolution. To get a more accurate timer, an attacker therefore needs to reduce the impact of the pseudo-random jitter. On both Chrome and Firefox, the jitter is deterministically defined by computing a random midpoint between the highest and lowest clamped values. However, the time of a jittered clock edge follows a triangle distribution around the precise timing value. This means that each measurement has a slight tendency towards the real value. By repeating the measurement several times, an attacker can, therefore, achieve a higher precision.

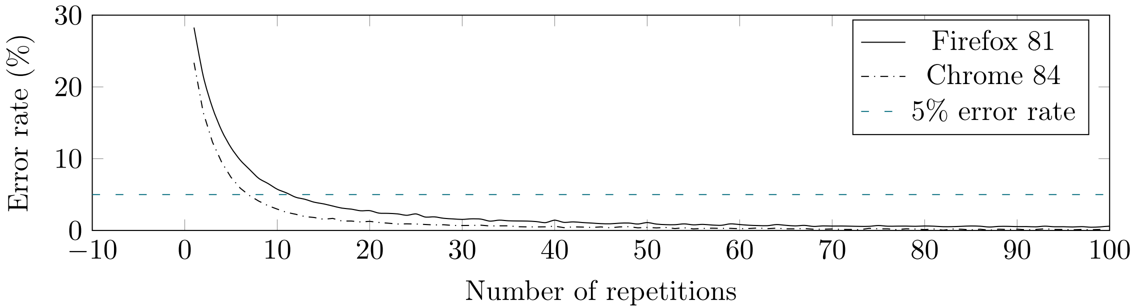


Figure 3.4. – Hit / miss error rate in function of repetitions, using `performance.now()` interpolation on Firefox 81 without COOP/COEP and Chrome 84.

Resolution To evaluate the impact of amplification on the resolution, we repeated the measurements and computed each time the average error rate on cache hit/miss discrimination. Figure 3.4 illustrates the results for Firefox 81 without COOP/COEP and Chrome 84. The error rate follows a logarithmic decrease with repetitions for both browsers. An attacker can reach a 5% error rate by repeating the measurements 15 times on Firefox 81 without COOP/COEP and 8 times for Chrome 84. On Firefox 81 with COOP, a mere 2 repetitions grant a 4% error rate. Comparatively, on an older version of Chrome, namely version 71, an attacker needed 12 repetitions in order to reach the 5% error rate.

Measurement overhead A single measurement without amplification takes at least the duration of a clock edge. That is, on average, 1 ms for Firefox 81 and 5 µs for Chrome 84. In the best scenario, repeating the measurement n times will increase the measurement time by n.

Specifically, repeating the measurement 15 times on Firefox 81 without COOP/COEP takes 29 million cycles on average. This represents a measurement overhead of 18 ms on our 1.60 GHz CPU. This means that building a covert channel using this timer as a receiver yields, at best, an ideal bit rate of 60 bit/s.

On Chrome 84, repeating the measurement 8 times grant a measurement overhead of 72 000 custom `performance.rdtsc()` cycles, or around 45 µs. This would grant an ideal bit rate of 22 kbit/s.

On Firefox 81 with COOP/COEP, with amplification, the measurement time would be 70 000 custom `performance.rdtsc()` cycles, granting as well an ideal bit rate of 22 kbit/s.

Conclusion The introduction of non-timer-based countermeasures, specifically site isolation for Chrome and COOP/COEP for Firefox have led browser vendors to take a step backward on timer-based countermeasures. Lowering the resolution and the jitter has an impact on the measurement overhead, hence on real-world attack time. Starting from Chrome 72, the resolution of `performance.now()` has been set to 5 µs with jitter. Starting from Firefox 79, when COOP/COEP are set, the resolution of `performance.now()` is set to 20 µs without jitter.

A cache covert channel based on `performance.now()` amplification on Firefox 81 with COOP/COEP has an ideal bit rate 360 times higher than without COOP/COEP. Similarly, an attacker with COOP/COEP can build an eviction set in around a few seconds, where an attacker without COOP/COEP would need several minutes. The same goes with Chrome:

on version 71, with a resolution of 100 µs, we obtain, with amplification, an ideal bit rate of 800 bit/s, 30 times lower than with the new Chrome 84 timer values. An attacker using Chrome 84 can build an eviction set in a matter of seconds, as opposed to several hundred seconds for Chrome 71.

The efficiency of the jitter is also highlighted by these results: between Chrome 84, with jitter, and Firefox 81 with COOP/COEP, without jitter, the measurement overhead is similar, when Chrome offers a better base resolution.

These changes have a massive impact on a browser-based threat model, where a script running with high performance for several minutes is highly suspicious, and can be detected by browsers. In a first-party scenario, an attacker can easily setup COOP/COEP and use a powerful timer granted by this change of resolution and timer. For both browsers, the impact of this change in resolution is a massive increase in the threat of timing attacks, as it means that P1 is less mitigated than on older versions. Changing the timer resolution does not prevent attacks, but impacts massively the time needed to execute them, which is an important factor in web security, as the user has to stay on the malicious web page for the duration of the attack. Setting a lower resolution and lower jitter has a double impact on the measurement overhead: as each clock edge is shorter, each measurement is faster. The attacker therefore needs fewer repetitions to eliminate the jitter, furthermore accelerating attacks.

3.4.3. Longitudinal study of `SharedArrayBuffer`-based clocks

Resolution The resolution of `SharedArrayBuffer` only depends on the computation time of an increment of the shared value. The smaller the value, the higher the resolution. We used `performance.rdtsc()` before and after incrementing the array buffer and tested two different types of increments: a simple increment (`value++`) and the built-in method `Atomics.add` [ECMa]. The Atomics API offers methods to safely use shared memory and handle conflicts.

We observed that a simple increment takes in the order of 20 custom cycles on both Chrome 84 and Firefox 81 whereas using Atomics takes 100. Logically, handling concurrent accesses introduces an overhead. We also noted that the first increment is slower, by an order of magnitude, probably from a cache impact. As the sub-thread systematically increases the same value, we excluded it and focused on the following values. `SharedArrayBuffer` offers a resolution in the order of 20 CPU cycles, or 10 ns on our CPU.

Measurement overhead The measurement overhead for `SharedArrayBuffer` is roughly the time it takes to read a value in the shared array twice: before and after the event. We again used our custom `performance.rdtsc()` to measure this time, and we tested two methods: standard array access and `Atomics.load`. On Firefox 81, a standard access lasts in the order of 42 custom cycles, opposed to 160 with Atomics API. As the lowest measurement overhead is preferable, we used the standard access. The measurement overhead on Chrome 84 with the standard access is 40 `performance.rdtsc()` cycles, or 20 ns on a 1.60 GHz CPU. It is similar on Firefox 81.

Conclusions `SharedArrayBuffer` have been disabled by default in Chrome 60 and Firefox 57.0.4 to mitigate Spectre. With the introduction of mitigations to transient execution attacks, they have been reimplemented. They are available by default in Firefox 79 with COOP/ COEP, and by default in Chrome 68.

SharedArrayBuffer based timers are, by far, the most powerful timer available in browsers. Table 3.2 illustrates the resolution and measurement time for SharedArrayBuffer-based clocks. They offer a resolution of 20 cycles and a measurement overhead of 40 cycles, equivalent to 10 ns and 20 ns respectively on a 1.60 GHz CPU. The offered resolution is sufficient to implement all known timing attacks. In addition, they have a very low measurement overhead and do not need amplification. An attacker using SharedArrayBuffer to build a covert channel can achieve an ideal bit rate of 50 Mbit/s on both browsers. This is 800 000 times higher than with performance.now() on Firefox 81 without COOP/COEP, and 2000 times higher than Chrome 84 and Firefox 81 with COOP/COEP.

An attacker could theoretically create an eviction set in less than a tenth of a millisecond by using this method. However, the algorithm required to create an eviction set has other sources of heavy computation, and still run within a few hundred milliseconds on our system.

Free access to such a powerful timer shows that P1 is not mitigated. Excluding Spectre-PHT, most of state-of-the-art attacks are theoretically possible under the current state of Chrome and Firefox, as P1, P2 and P3 are not mitigated, and P4 only partially mitigated. Other transient execution attacks, such as RIDL [vSMÖ+19] or ret2spec [MR18] are not prevented by COOP/COEP or site isolation, and are still implementable under certain conditions as the shared system resources are still accessible. In Firefox, where SharedArrayBuffer are restricted to sites with COOP/COEP, an attacker can still use them in a first-party scenario. Logically, countermeasures on performance.now() or other timers are secondary when SharedArrayBuffer are available, because they allow the creation of way more potent timers.

3.5. Discussion

In this section, we discuss the current state of timers in browsers and the challenges surrounding them.

Usability vs security and the lack of proper mitigations In Section 3.2, we have seen that the behavior of API timers like performance.now() has varied over the years in response to newly discovered attacks. Timing-based countermeasures (C2) have been widely implemented, but to various degrees of strength. On one hand, browser vendors decreased timer's resolution and added jitter to provide a more secure environment for their users. But on the other hand, their decisions appear arbitrary in retrospect as changes to timers were made without any concrete evidence of their effectiveness. For example, despite the study of Oren et al. in 2015 [OKSK15], it was not until 2018 that we saw the first implementation of jitter in web browsers. In that time frame, the decrease in timer resolution could simply be bypassed through interpolation [SMGM17]. Moreover, since some genuine applications are directly impacted by the lack of real-time precision provided by the affected timers [Bug18c], each browser vendor has tried to balance security with usability: Chrome never went above 100 µs, Firefox hovered around the 1 ms mark and got down recently to 20 µs while the Tor Browser has kept a 100 ms resolution since April 2015. The same goes for SharedArrayBuffer: they are enabled by default on Chrome, Edge and Opera, enabled under COOP/COEP for Firefox, and disabled on Safari and Tor.

This lack of consensus between vendors highlights how uncertain the industry is with the provided fixes. The results provided in this chapter show that the current timer-based countermeasures (C2) are a good first step towards protecting users but they still fall short of

fully protecting them against a large range of timing attacks, as P1 is a shared prerequisite between most of the timing attacks, and allegedly future timing attacks.

An alternative that could be considered by vendors is to put access to a high-resolution timer, based on `performance.now()` or `SharedArrayBuffer`, behind a permission. This way, when a developer needs it for an application or a game, she would need to ask the user for an explicit permission. This would prevent stealthy usage of API timers for timing attacks and all vendors could adopt the exact same very low resolution by default as it would not break pages not needing it.

The false sense of security created by resource isolation Recent trends on the development of the web as a platform have focused a lot on controlling what is running on a web page. Isolation-based countermeasures (C1) are at the core of browser security. Mechanisms like SRI [Conh], CORP [Conc], COOP/COEP [AJ], site isolation [RMO19] or even a proposal for better cookies [Wes] are all pushing the web forward in strengthening security. Yet, when it comes to timing attacks, all these new barriers create a false sense of security even though some attacks are definitely now much harder to pull off than before. By design, these countermeasures are not meant to mitigate P2 nor P3, and only a subset of P4. They are a great step forward in terms of security, but are not sufficient alone to mitigate the vast majority of timing attacks. While they greatly limit the possibility of a third-party attack when everything is set up properly on a web page, an attacker can still host her own malicious domain and conduct the attack from there. Moreover, the recent increase of timer resolution coupled with the reactivation of `SharedArrayBuffer` represents a massive step back in security where some attacks can be run in similar conditions to the ones in 2015. Our study highlights the dangers of coming back to such a state, and we hope browser vendors will recognize their mistakes by considering stronger mitigations to P1.

The need to mitigate timing attacks at the OS or hardware level Since timers can represent such an important threat to the security in a web browser, one can wonder if it would be possible to have a browser without timers. One approach is to try and make the browser deterministic as prototyped by Cao et al. with DeterFox [CCLW17]. By transforming a physical clock into a logical one, they change the behavior of known timers so that they do not return the time a request takes but return the number of operations that are being executed. While promising, this approach presents several shortcomings. First, they have to patch each known clock individually to instill this new behavior. A deep re-engineering of the browser would have to be made to possibly cover all implicit clocks. The second problem is that a logical clock loses all its meaning in the context of a real-time application. Some programs need to access the actual physical clock of the system to function properly and having a deterministic clock would present a lot of hurdles for developers.

In the end, we believe that everything running in a web browser has the potential to be a timer. This means that fully mitigating P1 seems out of reach. While browsers in 1995 mainly rendered static pages, the web has kept growing since then and it is now this rich and dynamic platform that can not only render pages but it is also the home of real-time communications and virtual reality, to name but a few. While some alternatives like putting access to high-resolution timers behind a permission can improve security, we simply have to learn to live with timers as they are such an intricate but integral part of the web.

As a consequence, mitigating timing attacks at the browser level is not the only solution

as we can develop solutions at both the OS and hardware level to provide stronger security against such threats, particularly mitigating P2 and P3. Software mitigations to transient execution still have, at this point in time, a high cost in performance. Browser vendors claim that mitigation must come from a lower level than software [TS] as hardware and transient execution attacks originate from microarchitectural optimizations.

3.6. Conclusion

We have studied the evolution in the last years and the current state of JavaScript-based timers and timing attacks for Chrome and Firefox, evaluating the resolution and measurement overhead for the two most efficient timers: `performance.now()` and `SharedArrayBuffer`. Timer-based countermeasures like clamping the resolution and adding jitter do not prevent attacks, but increase the time needed to exploit these attacks. Unlike in native environments, exploitation time is an important factor in web-based attacks, where the victim may not stay on the same web page for more than a few seconds or minutes. Unfortunately, the current trend of browser vendors undermining previous timer-based countermeasures by re-enabling `SharedArrayBuffer` and increasing the resolution to improve usability re-opens the door to many practical timing attacks, that were thought to be mitigated years ago. For example, the reintroduction of `SharedArrayBuffer` on Chrome brought a 2 000-fold increase in covert channel capacity, compared to `performance.now()` alone. This is even more dramatic on Firefox, where the increase is 800 000-fold. Powerful countermeasures such as site isolation and COOP/COEP only prevent a sub-class of transient execution attacks, thus browsers are currently vulnerable to other transient execution attacks, as well as a large range of timing attacks, with a large threat surface.

3.7. Evolution of Timer Security Since The Publication of the Results

The original work from this contribution dates from November 2020. Since its publication, the state of the art on timing attacks and their countermeasures has evolved. In particular, starting from Chrome 92 [Gooc], site isolation is no longer sufficient to retrieve `performance.now()`'s 5 µs and `SharedArrayBuffer`, but rather requires COOP/COEP to be enabled, using the same system as Firefox. When COOP/COEP is not enabled, `SharedArrayBuffer` are disabled and `performance.now()` has a resolution of 100 µs with jitter. This decision allows to reduce the threat models of timing attacks, as COOP/COEP introduces more security against, for instance, third-party attacks. However, the debate on security against usability is still present. In particular, the adoption of COOP/COEP is long and can be complicated to implement for websites with a large number of origins. To that extent, changes to the origin policy brought by COOP/COEP are currently being discussed [Vah], such as easing the policy regarding loaded resources, or allowing `SharedArrayBuffer` and a higher resolution on `performance.now()` without COOP/COEP when site isolation is deployed, i.e., in latest versions of Chrome and Firefox.

4 Port Contention in the Browser

In 2018, port contention attacks have been shown to be a potential attack vector in a technique introduced by Aldaya et al. [ABuH+19], named PortSmash. This attack on Intel CPUs is based on port contention, where CPU ports act as a bottleneck in the execution pipeline. By sharing ports with the victim, the attacker can exploit timing differences caused by the contention of different instructions. PortSmash has a high temporal resolution and can be used, like its counterparts on the cache, to perform side-channel attacks on cryptographic libraries. While port contention attacks restrict the attacker by requiring that it shares the core it executes on with its victim, they are inherently stealthier than attacks on the memory subsystem. They are also immune to most hardware and system countermeasures which, in their vast majority, target the cache [PGGV21, DFS20, LL14, KASZ09, ZR13, SQ21].

When compared to cache attacks such as Prime+Probe, native port contention attacks offer better speed and spatial accuracy, do not require a complex cache profiling step, are more resistant to noise, and, most significantly, can bypass cache-centric countermeasures. Mounting a port contention attack in a browser setting would therefore deliver a real advantage to attackers. Performing such an attack, however, is far from trivial. The basic step of a Prime+Probe cache attack is sequential access to user-controlled memory. It has been shown that even high-level primitives, such as substring searches, can provide this functionality [SAO+21]. Port contention, on the other hand, requires an attacker process that is co-located with the victim on the same processor core and executes assembly language instructions carefully chosen to conflict with the victim's instructions. This is highly challenging in a web browser environment:

C1 In this setting, the attacker's code is written in a highly-abstracted language which is translated into machine code by a just-in-time compiler;

C2 The attacker has no control over the physical core selected by the browser to execute the attack code;

C3 Finally, web-based timers have a lower resolution than native hardware-based timers, increasing the attacker's measurement noise.

This chapter tackles these challenges, and asks the following questions: Can port contention attacks be mounted from within the browser? What are the implications of this new attack vector?

Contributions This chapter makes the following contributions:

[1] This work was presented before heavy countermeasures against timing attacks. The covert channel is theoretically still implementable, but with a heavily degraded bandwidth.

Table 4.1. – Comparison of covert channels in web browsers.

Covert channel	Bandwidth	Runs with current mitigations	Setup
CPU throttling [RRR16]	0.2 bit/s	●	-
Disk contention [vGJ17]	5 bit/s	●	-
RIDL (Evict+Reload) [vSMÖ+19]	8 bit/s	●	-
DRAM [SMGM17]	11 bit/s	●	-
Hardware interrupts [LGS+17]	25 bit/s	●	cross-browser
Event loop [VK17]	200 bit/s	○	cross-browser
Prime+Probe [OKSK15]	320 kbit/s[1]	◐	
Prime+Probe [OKSK15]	8 kbit/s[1]	◐	cross-VM
Port contention [our work]	200 bit/s	●	**cross-browser**
Port contention [our work]	80 bit/s	●	**cross-VM**

- We show that port contention can be ported to web browsers via WebAssembly, despite the strong requirements of this attack and the abstraction of the WebAssembly language. This greatly increases the attack surface that is due to port contention (Section 4.1).

- We propose an automated framework to find which WebAssembly instructions can cause port contention on a given Intel processor (Section 4.2).

- We demonstrate a side-channel attack on a synthetic example, to evaluate the resolution of our port contention attack. We show that our attack has a spatial resolution of 1024 instructions with a single trace, of the same order of magnitude as the best microarchitectural attacks in the browser (Section 4.3).

- We build a covert channel using port contention. With a sender running unprivileged native code and a receiver inside the browser, we obtain a throughput of 200 bit/s, i.e., one order of magnitude higher than modern covert channels in the browser. Table 4.1 compares the results of our covert channel with the state of the art. In a virtualized setting where the sender is running inside a virtual machine, we reach a throughput of 80 bit/s. We also build a cross-browser covert channel with an estimated throughput of 200 bit/s. (Section 4.4).

4.1. Web-Assembly-Based Port Contention

We introduce, to the best of our knowledge, the first implementation of port contention inside a browser. We can create and measure port contention from the JavaScript sandbox, on both Mozilla Firefox and Google Chrome. We found instructions that create contention on both P1 and P5, allowing diverse potential victims.

Experimental setup and threat model Unless stated otherwise, we run all experiments on an Intel i5-8365U CPU with a maximal frequency of 1.60 GHz running Ubuntu 20.10, with

Mozilla Firefox 90 and Google Chrome 95 desktop version, both using WebAssembly 1.1[2]. As Safari and Edge support WebAssembly, the attack can theoretically be carried on these browsers, but they remain outside of the scope of this chapter. The threat model is similar to a user visiting a malicious website with his browser. The browser scripts run in a cross-origin isolated browser [Conb, Cona], granting more context isolation and allowing access to `SharedArrayBuffer` and higher resolution timers.

Description In principle, our web-based attack follows a similar attack flow than Aldaya et al.'s [ABuH+19] described in Section 2.4.1.4, but in the more restrictive JavaScript sandbox. The attacker is situated inside of the browser sandbox. During the attack, they repeat specific WebAssembly instructions that cause contention on a specific port. Section 4.2 explains how we find these instructions on different systems. For instance, on our processor, the WebAssembly `ctz` (Count Trailing Zeros) instruction creates contention on P1. Similarly, instructions that truncate floats to integers, e.g., `trunc_f32_u`, create contention on P5. The attacker then times the execution of these instructions. If no other processes use the same port at the same time, these instructions will all be executed in a row, resulting in a fast execution time. However, if another process emits μops on the same port, these μops will be queued with the attacker-generated μops, resulting in a slower execution time for the attacker. By measuring these differences in timings, the attacker process can monitor the port usage on a specific port, and thus monitor other processes.

Challenges We face three challenges when implementing port contention in the browser. First, as browser-based scripts run in a controlled sandbox, we have no access to native instructions, and must instead use higher-level language constructs (**C1**). Furthermore, as browser-based scripts are meant to be portable, the instructions are translated to different assembly language instructions by the browser's engine on different systems. This means that the same script generates different native instructions depending on the CPU architecture, each with a different port usage, varying from vendors and generations. The code is also highly optimized by the engines, and execution can vary even on the same system, based on the variables or structure of the code. To gain more control over the port usage of our attacks, we mounted our attack with WebAssembly. This grants us access to smaller, more atomic instructions. However, these instructions are still executed through the browser's JIT engine, and their translation to machine language can vary from one system to another. For instance, the WebAssembly instruction `ctz` is translated into the native Intel instruction `TZCNT` on our system, as we describe in more detail in Section 4.2. The `TZCNT` instruction, in turn, is implemented using a single μop which is executed on P1 [AR]. Thus, repeatedly executing the WebAssembly instruction `ctz` can cause contention on P1. The Intel instruction `TZCNT` is only available, however, on CPUs starting from the Broadwell generation. Thus, the WebAssembly `ctz` instruction may generate contention on another port in older CPU generations. Directly compiling native code using x86 assembly instructions to create contention is not possible. Since WebAssembly is designed as a portable language, the compilers cannot emit instructions that are directly architecture-dependent, as they could not run on non-Intel CPUs.

Secondly, the high level of abstraction provided by the browser means that an attacker can neither know nor control on which core the attack is executed (**C2**). Furthermore,

[2]We used the latest version available in November 2021. This version did not support vectorial types and SIMD instructions.

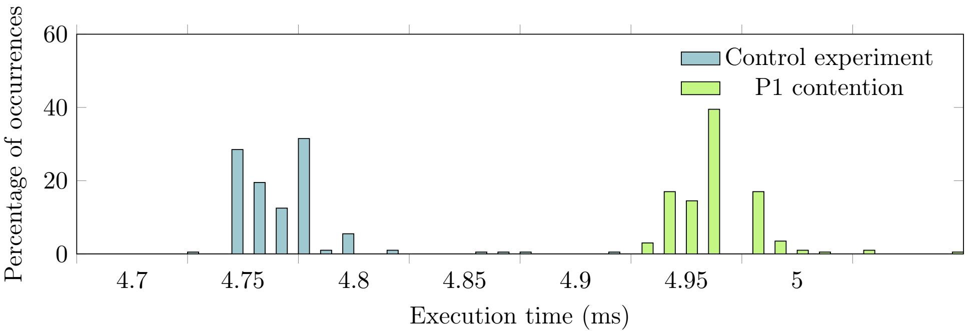


Figure 4.1. – Port 1 contention experiment on `i64.ctz` for 1 000 000 instructions.

the operating system's scheduler dynamically moves processes between cores to optimize computing and save energy. We address this challenge by performing our attack on multiple cores simultaneously by using Web Workers, JavaScript multi-threading implementation, which creates a sub-thread running in a different process. This lets the attacker create as many attacker processes as physical cores, and as they all have a high workload, they are spread on different physical cores. Then, one of the attacker processes runs on the same core as the victim process, able to monitor it.

Finally, our attack requires high-resolution timers to monitor processes at the μop level (**C3**). Native implementations of port contention attacks all use the cycle-accurate `rdtsc` instruction. As explained in Section 2.6.2 and Chapter 3, browser vendors have restricted access to such timers inside of the sandbox to prevent timing attacks, but auxiliary timers can still be built, letting attackers recover a high-resolution timer. In our attack, unless stated otherwise, we use `SharedArrayBuffer`-based timers, which offer a resolution and measurement time in the order of 20 ns [SMGM17, RML21].

Proof-of-concept Figure 4.1 shows a proof-of-concept illustrating the contention on P1 caused by the WebAssembly `i64.ctz` instruction.

In this experiment, we time the execution of 1 000 000 WebAssembly `i64.ctz` instructions using the low-resolution JavaScript function `performance.now`. We run the experiment on Firefox 90, where this timer offers a resolution of 20 µs without jitter. In parallel with the Firefox code, we also run a sender program written in native code and pinned to the same processor. In the P1 contention experiment, the native sender runs the Intel instruction `crc32` in a loop. This assembly language instruction is known to cause contention on P1. In the control experiment, the native sender runs a simple loop designed not to cause port contention. We run this program, instead of simply not executing the sender at all, to ensure that the difference stems from port contention, and not from other sources. As the figure shows, the timings measured during the P1 contention experiment are on average 5% higher than the control experiment, allowing the browser to efficiently distinguish between the two distributions. We observe similar results on Chrome 95.

In the following sections, we describe how to convert this proof-of-concept into practical attacks. In particular we obtain a higher spatial resolution and evaluate 100 WebAssembly instructions (**C1**), we ensure the attacker does not have to pin processes (**C2**), and we use a higher resolution timer (**C3**).

4.2. PC-detector

The translation of WebAssembly instructions into μops is variable on different systems: it can depend on the microarchitecture, instruction extension sets or JavaScript engine. In this context, it can be hard to find WebAssembly instructions that reliably cause port contention. In this section, we propose PC-detector, a Selenium-based framework to dynamically detect and characterize the port usage of WebAssembly instructions. Using the methodology described in Section 4.1, PC-detector automatically tests multiple WebAssembly instructions and checks if they cause contention on P1 or P5.

4.2.1. Description

Framework Our framework is composed of two components. The first component is a native C script that either runs an empty loop, creates contention on P1, or creates contention on P5. The second component is a Selenium-controlled browser which runs automatically generated WebAssembly code. For each WebAssembly instruction *instr*, we create a binary file with 1 000 000 calls. This file is then executed in the browser, and its runtime is measured using `performance.now()`[3]. We run three experiments:

1. Repeatedly executing and timing the WebAssembly file, used as a control.
2. Creating contention on P1 with native code and timing the WebAssembly file.
3. Creating contention on P5 with native code and timing the WebAssembly file.

By evaluating the timing distributions of these three experiments, we can evaluate the port usage of *instr*. If the three distributions are mixed, *instr* is not affected by the port contention (thus it cannot cause it). If the P1 timings (respectively P5) are, on average, higher than both the control and P5 (respectively P1), this means *instr* can detect, and cause, contention on P1 (respectively P5).

We evaluate all standardized single and double operand operations [Gro], including arithmetic operations and memory operations. Due to the stack machine structure of WebAssembly, each experiment includes a `load` operation to add values to the stack between each operation. We discovered that due to JIT optimizations, it is not possible to load many values on the stack before running double operand operations in a row, as the compiler reorders the instructions to alternate between loads and the tested operation. Therefore, we could not run all double operand operations one after the other. We evaluate single instructions when instructions have an output of the same type as their input, and pairs of complementary instructions in the other case (e.g., convert a 32 bit integer into a 64 bit float). We do not evaluate control flow operations, e.g., loops or jumps.

Metrics We propose two main metrics to automatically evaluate if a WebAssembly instruction can create contention on P1 or P5. The first one is based on the error rate between timings from the P1 and P5 experiments. For this metric, we compare P1 to P5 instead of P1 to control, as the control experiment does not run calculations on the native side. This

[3]Here, we use `performance.now()` instead of a `SharedArrayBuffer`-based clock as we control the number of calls to *instr*. This allows us to artificially augment the duration of the timed event to be measurable with `performance.now()`. This ensures us that contention is caused by our experiment, and not by noise created by `SharedArrayBuffer`'s increments.

means that the timing differences could originate from other sources than port contention, e.g., variation in frequency or contention on another shared hardware component. P1 and P5 have two timing distributions, and one distribution (X_{low}) has lower timings than the other distribution (X_{high}) when there is contention. Given a temporal threshold τ, we define the error rate as the proportion of values of $X_{low} > \tau$ and values of $X_{high} < \tau$ over all experiments. We define the error rate for a given threshold as

$$er_\tau = \frac{|X_{low} > \tau| + |X_{high} < \tau|}{|X_{low}| + |X_{high}|}.$$

Then, by computing er_τ for $[min(X_{low}) < \tau < max(X_{high})]$, we can retrieve the lowest error rate possible, giving us the probability for a program to blindly distinguish between port contention and standard usage from experiment timings. By inverting X_{low} and X_{high} and computing the best error rate, we can see if an instruction creates contention on P1, P5 or none. In PC-detector, we infer that if $er_\tau < 5\%$, an instruction creates contention.

The error rate calculation lets us identify whether an instruction creates contention. It does not, however, illustrate the efficiency of this contention, i.e., how separated both distributions are or how spread they are. This parameter is important in our attacks, as the more distance between the distributions, the easier it is to distinguish between contention and standard usage. In order to measure the distance between P1 and P5, we compute the effect size, also known as Cohen's d. In our case, Cohen's d between P1 and P5 is defined as

$$d = \frac{|mean(P1) - mean(P5)|}{\sqrt{(stdev(P1) + stdev(P5))/2}},$$

with stdev() the standard deviation of the distribution. A high Cohen's d means that distributions are highly separated and concentrated, and that we can more easily distinguish contention from standard usage.

4.2.2. Results

We have tested 100 different instructions, including numerical, memory, bit-wise, and type conversion operations.

Table 4.2 lists which instructions cause contention on the i5-8365U. The results are identical between Chrome and Firefox, although the distance varies because of the different browser architectures. In total, we found 21 instructions causing contention. As most instructions have 32- and 64-bit variants, some instructions are doubled. Generally, we observe that 64-bit variants have a greater Cohen's d than their 32-bit counterparts. Similarly, the unsigned variants of integer operations often grant better results than the signed variants.

P1 contention seems to be caused by arithmetic instructions, whereas conversion/truncation operations create contention on P5. This result is coherent with the specialization of ports and execution units. `i64.rem_u` shows the highest effect size of all detected instructions.

To demonstrate the portability of port contention and PC-detector, we have run the same benchmark on different Intel CPUs. In total, we have tested 4 recent CPUs: i5-8365U, i7-8650, i7-10510 and i7-10610. The instructions creating contention remain constant, but Cohen's d can vary based on the CPU frequency. This is logical, as all tested cores have the same instruction set extensions, meaning that the WebAssembly instructions are translated to the same native instructions.

Table 4.2. – WebAssembly instructions causing port contention. For clarity, we group together the 32- and 64- bits versions of instructions under one line marked i32/i64.

Instruction	P1 contention	P5 contention	Cohen's d
`i32/i64.ctz`	●	○	1.2
`i32/i64.clz`	●	○	1
`i32/i64.popcnt`	●	○	1
`i32/i64.div_s`	●	○	10
`i32/i64.div_u`	●	○	10
`i32/i64.rem_u`	●	○	34
`i32/i64.rem_s`	●	○	5
`f32.convert_i32_s and i32.trunc_f64_s`	○	●	1
`f32.convert_i32_s and i32.trunc_f32_s`	○	●	2
`f32.convert_i64_s and i64.trunc_f32_s`	○	●	8
`f32.convert_i32_u and i32.trunc_f32_u`	○	●	2
`f32.demote_f64 and f64.promote_f32`	○	●	3
`i32.wrap_i64 and i64.extend_i32_u`	●	○	16
`i32.wrap_i64 and i64.extend_i32_s`	●	○	11

4.3. Side-channel Attack on Artificial Applications

In this section, we present an artificial gadget, illustrating the side-channel threat of web-based port contention. We built a synthetic and generic example showing how a program, which execution depends on secret information, is vulnerable to WebAssembly port contention. Indeed, if a program has branches depending on secret bits, an attacker can use a side-channel attack to infer the secret. The victim process is an unprivileged native process. The attacker is a JavaScript and WebAssembly script, running inside of the browser's sandbox. The attacker has no access to addresses, native instructions, and no control or knowledge of physical or logical cores.

In our implementation, an attacker, running code inside the browser's sandbox, monitors the victim's execution with a spatial resolution of 1024 native instructions, i.e., 3072 bytes. This spatial resolution is of the same order of magnitude as other microarchitectural attacks in the browser, e.g., Prime+Probe, which has a resolution of a cache set (typically 12 to 20 cache lines), i.e., 1280 bytes on our system.

4.3.1. Description

The victim is a native unprivileged program, running different code sections based on the bits of secret information. As port usage differs between branches, an attacker monitoring port contention could infer parts of the secret. Figure 4.2 illustrates our gadget, implemented in native assembly code. Depending on a secret bit, the code will execute either instruction creating contention on P1 or P5. To detect from within the browser which path is taken by the victim, we time the execution of nb_{instr} WebAssembly `rem_u` instructions, which creates contention on P1 (Section 4.2). If the execution time is high, then we know that the native code also creates contention on P1, whereas if it is standard, we know that the native code

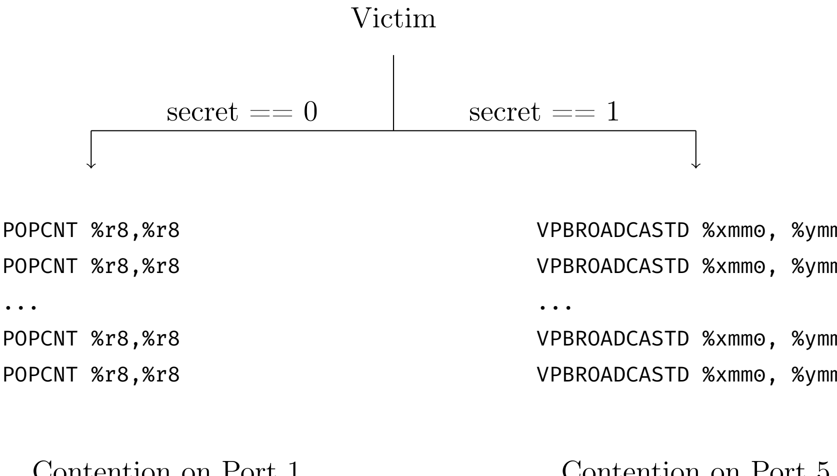


Figure 4.2. – Side channel artificial example. Depending on the key bit passed in parameter, the code will have different port usage.

does not create contention. By repeating this process, we detect the branch that was executed by the native script, and hence the value of the secret bit.

After resolving **C1** with PC-detector and **C3** with `SharedArrayBuffers`, we still face the inability to pin the attack code to the same physical core as the victim (**C2**). Most schedulers try to balance the workload between physical cores. By creating a number of listening Web Workers equal to the number of physical cores, we maximize our chances that one of them listens on the victim's physical core, thus circumventing **C2**. Information about the system's core count is available through the `navigator.hardwareConcurrency` JavaScript API [Cone], available by default on both Chrome and Firefox.

4.3.2. Results

An important metric for our evaluation is the spatial resolution, i.e., the smallest number of instructions we can detect in a branch. To detect contention, we measure the execution time of nb_{instr} WebAssembly `rem_u` instructions. This parameter is important: a high number of instructions lowers our spatial resolution, but a lower number yields noisier time measurements. Furthermore, for values of nb_{instr} ranging from 1 to 10, the execution time of the instruction is slower than the read access to the shared array and other overhead introduced by JavaScript. This means that contention is measured at only specific times in the measurement. To reduce the measurement time of `SharedArrayBuffer`, we access the array directly, without using concurrent access libraries. This grants a better resolution to the timer but creates more noise and outliers. On our system, we were able to create a web listener running in the same physical core as the victim in 95% of our experiments. We infer that the remaining errors stem from the scheduler moving our process to different cores because of other threads creating noise.

On our system, we found $nb_{instr} = 10$ to be the best compromise between noise and resolution. To reduce the noise, we process the data with a median sliding window with a width of 10 measurements. Figure 4.3 illustrates the resulting values when the victim runs the code with the secret 1101001, for a single trace of the victim. The high values represents

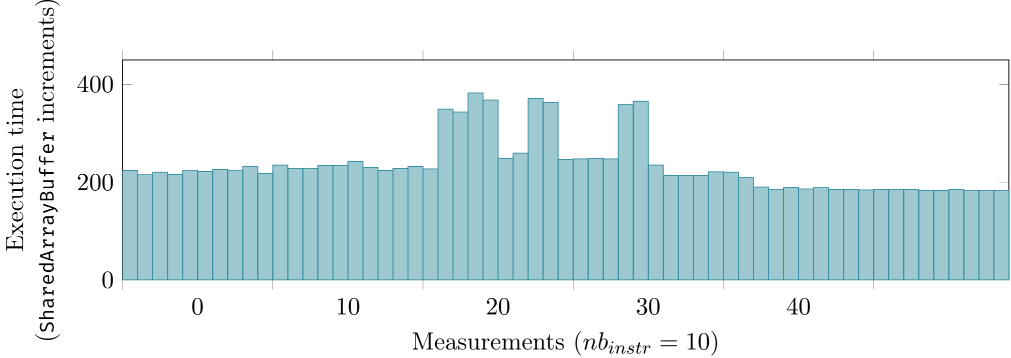


Figure 4.3. – Single-trace execution with secret information 1101001, on Chrome 95.

the execution of the victim branch creating contention on P1 i.e., a bit set to 1. The width of a peak or a pit is proportional to the number of bits inside the sequence.

Our implementation is able to detect the executed branch with a resolution of 1024 native instructions on both Google Chrome and Mozilla Firefox. To obtain this result, we first implement Figure 4.2 with a very high number of `POPCNT` and `VPBROADCASTD` instructions, that we progressively lower. The resolution is the lowest number of instructions where we can clearly retrieve the secret bits without error on a single victim trace.

This experimental limit of 1024 instructions is mainly due to the lack of access to high-resolution timers. Note that we observe two peaks per secret bit with a single trace. We have found that a higher resolution of 512 instructions could introduce errors with a single-trace attack. One solution to increase the resolution would be to revert to multiple-trace attacks. Moreover, by using a custom browser implementing `rdtsc`, based on the native cycle accurate timer, we observed that our implementation has a resolution of 256 instructions, i.e., a better spatial resolution than Prime+Probe. This means that our experimental limit could be lowered with better auxiliary timers or noise filters, which could offer a more fine-grained attack vector than existing microarchitectural side channels in the browser.

4.4. Covert Channel

In this section, we present a port contention-based covert channel with a throughput of 200 bit/s for a 1% error rate. This covert channel is composed of a sender running unprivileged native code, and a receiver running completely inside the browser (similarly as Schwarz et al. [SMGM17]). We also show that our covert channel runs with a sender located inside a VM, and can even be used in a cross-browser fashion (similarly as Lipp et al. [LGS+17]).

The sender runs unprivileged C code on the victim's hardware. The sender can therefore freely use most native instructions, and has access to cycle-accurate timers. It can also pin itself to a certain physical or logical core. The receiver, on the other hand, runs fully inside a cross-origin isolated web page. As it runs inside the browser's sandbox, the receiver has no access to native instructions. Port contention must be created and measured by using WebAssembly (**C1**). Moreover, the web script must share a physical core with the sender, but cannot control or know on which physical core it is running (**C2**). Finally, the receiver does not have access to high-resolution timers (**C3**). Instead, we use `SharedArrayBuffers` to get the best resolution available.

4.4.1. Description

We implemented a half-duplex asynchronous channel based on port contention, between a native sender and a web-based receiver. In addition to data, the sender and receiver exchange control messages to handle acknowledgments and synchronization. Both parties must therefore be able to send and receive bits. Our side channel can be decomposed into two layers. The lower layer, sending and receiving bits, is equivalent to the physical layer of the TCP/IP model. This layer uses CPU ports as its transmitting channel, and must be able to distinguish between 0 and 1 bits. The upper layer is equivalent to the data-link layer. This layer handles the synchronization between the parties, as well as error management.

Physical layer The two parties send 1-bits by creating contention on P1 for a fixed duration (t_{bit}), and send 0-bits by idling for t_{bit}. t_{bit} is an important factor, as a high duration lowers the channel's bandwidth but allows the receiver to tolerate more noise when attempting to distinguish bits. In our covert channel implementation, we have fixed $t_{bit} = 1\,\text{ms}$. To create contention, the sender and receiver repeatedly call an instruction, respectively the native Intel instruction `crc32` and the WebAssembly instruction `rem_u`. To receive a bit, the sender or receiver repeatedly calls these instructions while timing them. A high execution time means the emitting party is sending a 1, while a standard time means a 0. As both instructions are handled by the CPU port P1, both the sender and receiver cannot emit at the same time, making our channel a half-duplex channel. Besides their high resolution, another advantage of a `SharedArrayBuffer`-based timer is that it is based on a Web Worker, and therefore runs on a different core. This lowers potential noise on the covert channel.

We also need to ensure both the sender and receiver are running on the same core (**C2**). As the browser cannot control which core it is running on, the sender creates as many sub-senders as physical cores. The sender runs unprivileged native code, so it knows the number of physical and logical cores, and can pin each of its sub-threads to a specific core. This ensures that at least one sender thread is running on the same physical core as the receiver.

Although `SharedArrayBuffers` offer a high resolution, they can introduce errors at the physical layer level. In particular, concurrent accesses between the thread incrementing a value and the main thread reading the timestamp can cause insertion or deletion errors. We have determined two error-prone scenarios at the physical level. In the first scenario, the main thread reads the shared value too frequently. This prevents the clock thread from incrementing the value, and as a result the measured time is much lower than the real-time value. The other scenario stems from particularly high measurement outliers when contention is created. We assume it also comes from concurrent accesses. As this access is longer than usual, it means that we can get less measurements during t_{bit}, thus creating bit deletion errors on higher layers.

Protocol and frame format To ensure synchronization and correct potential errors, we implemented a simple protocol above our physical layer, similarly as Maurice et al. [MWS+17]. Figure 4.4 illustrates a typical exchange, as well as packet loss management. It is based on a simple request-to-send scheme: the receiver sends a request frame (described in Figure 4.5a), containing a 4-bit sequence number. Upon reception, the sender sends a data frame (described in Figure 4.5b), containing the sequence number as well as the associated data (1 byte). If the data frame is received correctly, the receiver requests the next sequence number. Both

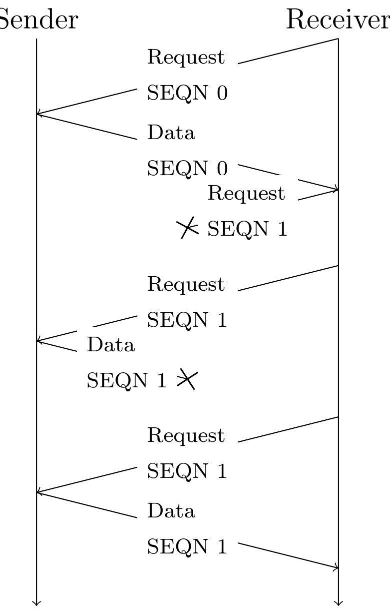


Figure 4.4. – Illustration of the protocol's synchronization in case of lost or incorrect packet.

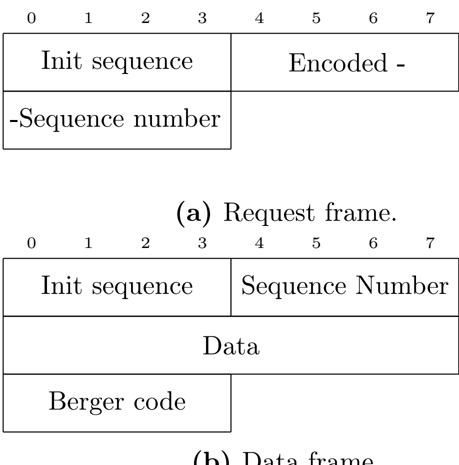
frames start with a 4-bit preamble consisting of an initial sequence which is always set to 1010. This initial sequence serves as calibration for the receiver.

To handle possible insertion or deletion errors, we added an error detection code. More specifically, the sequence number is encoded with (8,4) Hamming code [Ham50] in request frames, and the last 4 bits of the data frame contain a Berger code [Ber61], counting the number of zeros in the data and sequence number fields. As the type of errors we face are mainly bit insertion or deletion, we do not use the error correcting properties of Hamming code, and instead use it as an error-detection code.

Our protocol encodes 8 bits of payload into a 31-bit message, including the preamble, sequence numbers and error detecting code. This means that, with t_{bit} =1 ms, we can reach a maximal raw throughput of 1 kbit/s, i.e., a theoretical maximum of data bit rate of 260 bit/s.

Our protocol also manages packet loss and desynchronization. This is handled by the sequence number and the request-to-send scheme. As illustrated in Figure 4.4, after sending a request, the receiver waits for a fixed timeout value. If it has not received an answer at the end of this time period, it simply re-sends the request. This lets the covert channel recover from packet loss from the sender to the receiver, and from the receiver to the sender.

Receiving frames The sender and receiver do not share a common clock. Hence, the party receiving bits does not know in advance the demarcations between successive bits, nor when the frame starts. It is processing execution time of instructions as a real-time stream of information, not in post-processing. In order to automatically detect the start of the frame, as well as the actual bits, both sender and receiver run DenStream [CEQZ06], a density-based data-stream clustering algorithm. It dynamically creates clusters of data, based on the execution time and their time of arrival. The listening party then detects the start of the frame when it detects 4 consequent small clusters with variation in execution time, corresponding to the initial sequence of 1010. The initial sequence is used to calibrate two



(a) Request frame.

(b) Data frame.

Figure 4.5. – Format of the request and data frames.

major values: the temporal threshold between 0-bits and 1-bits, as well as the average number of instructions in a single bit. The average number of points lets the algorithm detect the number of bits in a sequence. As DenStream computation can be slow when we reach a high bit rate, we only use it to detect the preamble. For the rest of the frame, we use a simple stream-based threshold detection: timings above the calibration threshold are identified as 1-bits, and others as 0-bits.

To infer the actual number of bits in such a sequence, we use the average number of instructions calibrated from the initialization sequence. Then, by dividing the number of instructions in our same bit sequence, we can infer how many bits it contains. This step is prone to insertion or deletion errors.

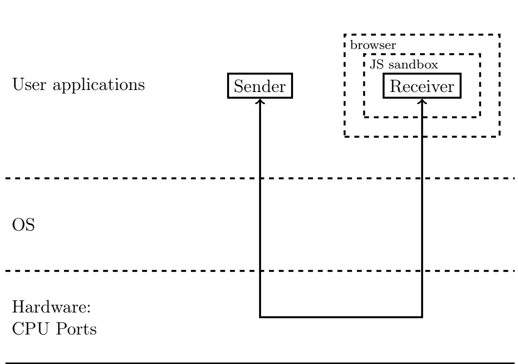
When the stream algorithm has detected a number of bits corresponding to the frame size, it stops listening. If the frame is invalid because of insertion and deletion errors, we try to reinterpret it with slightly modified calibration values. Indeed, variation in frequency can cause slight changes on the number of measurements in a bit, e.g., a frequency raise means we measure more instructions in a bit, thus potential insertion errors.

4.4.2. Evaluation

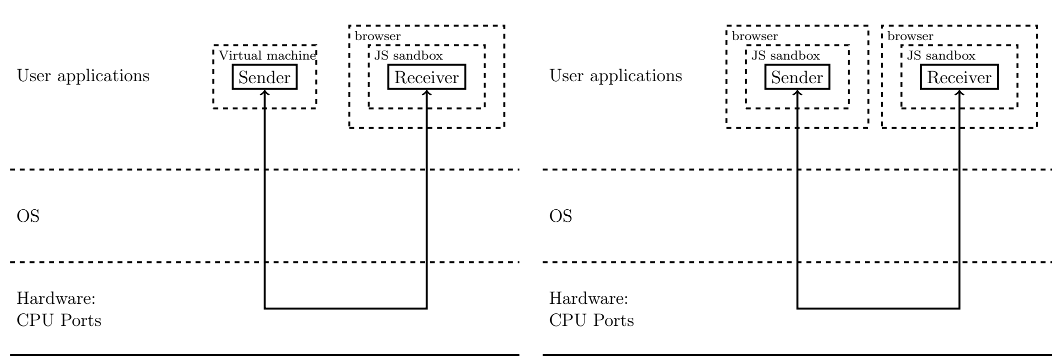
We evaluated our covert channel in two different scenarios. The first scenario is the baseline implementation, where both the native sender and web-based receiver run in a standard OS. In the second scenario, the native sender now runs in a virtual machine running on the victim's physical hardware, while the browser runs in the standard OS. This scenario is common, as malware analysis is often conducted in sandboxed environments such as VMs. We also evaluate the impact of noise on our covert channel.

Native sender This threat model represents the most common scenario, where both the browser and the native sender run as unprivileged processes in the OS. We evaluated our covert channel by transmitting 10 kB of data from the native sender to the web-based receiver. To compute the error rate, we compare the original and received bit sequences bit-by-bit.

Table 4.3 illustrates the bit rate and error rate of our channel in different noise conditions. The transmission takes, on average, slightly less than 7 min. During the transmission, on



(a) Standard native-to-web scenario



(b) VM-to-Host scenario

(c) Cross-Browser scenario

average 600 frames arrive incorrectly or are lost from the sender to the receiver, over a total of 10 600 frames. This represents a total frame loss rate of 5.5%. Most of the incorrect frames were the result of insertion or deletion errors. The lost frame rate from the receiver to the sender is negligible. We achieve a bit rate of 200 bit/s. This is 80% of the maximal bandwidth possible when using t_{bit} =1 ms. The difference between the bit rate upper bound and our implementation stems from the loss of frames, which requires the sender to wait for some time before requesting the data again, as well as from the short computation time required to handle the protocol.

In this setup, our covert channel presents a better bit rate than previous web-based covert channels [SMGM17, LGS+17, RRR16, VK17, vGJ17]. The only covert channel with a better resolution is Prime+Probe by Oren et al. [OKSK15]. However, recent countermeasures had a substantial negative impact on the bit rate. To the best of our knowledge, no other Prime+Probe covert channel has been implemented since that allows us to compare between the two approaches. The closest cache covert channel is the one presented by van Schaik et al. in RIDL [vSMÖ+19], with a bit rate of 8 bit/s.

We now evaluate our covert channel in the presence of noise. Noise can impact both the bit transmission through port contention, and the `SharedArrayBuffer` clock. Indeed, we observe that when stressing the physical core used by the `SharedArrayBuffer` clock, the number of ticks we measure in each time period decreases, in turn decreasing our resolution. However, our covert channel shows strong resilience to sources of noise with a low thread count. That is

Table 4.3. – Evaluation of the port-contention covert channel in different conditions.

Experimental setup	Bit rate	Packet Loss rate	Error rate
Noiseless	200 bit/s	5.5%	1%
`stress -c 2`	170 bit/s	8%	3%
`stress -m 2`	120 bit/s	15%	3%
`stress -c/-m 3`	25 bit/s	80%	5%
`stress -c/-m 8`	<1 bit/s	99%	5%

because port contention depends on the physical core. As our sender and receiver already use a major part of the core computing capacities, the OS scheduler tends to move other noisy processes to different physical cores, thus lowering their impact on our covert channel. For instance, when running `stress` with square root (`-s`) or malloc (`-m`) on two threads, the bit rate remains on the same order of magnitude. The loss of performance stems from a higher rate of lost frames due to clock outliers. Our channel also shows better resilience to sources of noise with a low thread count than cache covert channels, as the LLC is shared between cores [MWS+17]. However, if a noisy thread runs on a physical core shared either by the clock or the receiver, the performance significantly drops, as illustrated in the `stress -c 3` case. In that case, the lost frame rate increases drastically because of lower resolution from our timer. Introducing specific error-correcting codes to correct insertion or deletion errors could greatly improve the performance of the channel in noisy conditions.

Virtualized sender We also evaluate our channel in a virtualized setup. In this scenario, the native sender runs inside of a virtual machine running Ubuntu. The browser runs in the standard OS. The main change in the threat model is that the native sender has no control or knowledge of cores, physical or logical. However, by creating multiple sender threads and not pinning them, we managed to force at least one sender thread to run on a physical core shared with a receiver. In this setup, our covert channel has a bit rate of 80 bit/s. This bit rate is still higher than that of many browser covert channels [VK17, SMGM17, LGS+17, vGJ17, RRR16], and even equivalent to some native covert channels in the same setup [SMAK20].

4.4.3. Cross-Browser Covert Channel Bandwidth Estimation

Our covert channel can be extended to a cross-browser setup. As we can create and detect contention on the browser, we can replace the native sender with a JavaScript sender. This has two major impacts on the effectiveness of the attack. First off, the web-based sender loses access to powerful native timers, potentially creating new errors on the request frames. Most importantly, the browser has no knowledge of physical or logical cores. It cannot know nor control on which core it is running. To circumvent this difficulty, the web-based sender creates a number of Web Workers equal to the number of physical cores of the machine. By doing so, the scheduler will spread these senders on different physical cores. When launching the receiver, however, the senders are not the only processes using a high workload, and we have noticed that launching the receiver and the `SharedArrayBuffer` clock after the sender results in a physical core running both the clock and the receiver, and the senders sharing the remaining core. We overcome this limitation by initializing the clock and receiver before the

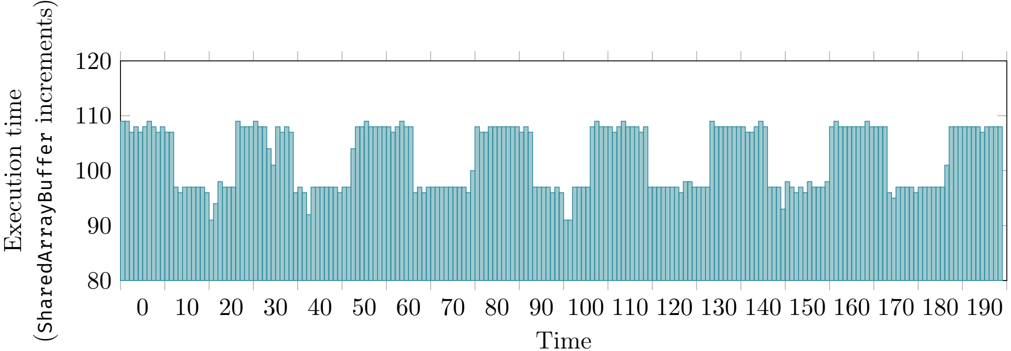


Figure 4.7. – Transmitted square signal from Firefox 90 to Chrome 94 with t_{bit} =1 ms

sender. As a result, the scheduler assigns a physical core shared by a receiver and a sender, effectively allowing the implementation of our covert channel.

Using this technique, we were able to transmit bits of information across browsers through port contention with t_{bit} = 1 ms. i.e., conditions equivalent to the native-to-web covert channel. We were able to demonstrate data transfer at the physical layer from Chrome to Firefox, from Firefox to Chrome, and between two instances of the same browser. Figure 4.7 shows the transmitted square signal from Firefox to Chrome. We did not re-implement the data-link layer to this threat model, as it represents significant engineering work, and leave it to future work. However, this proof of concept solves all scientific and technical challenges, including the most difficult, i.e., core management (**C2**), by its ability to transmit bits. As the physical layer offers similar bit and error rates to the native sender, even for a long duration of transmission, it is safe to estimate that this cross-browser covert channel can reach a final bandwidth on-par with the native-to-web covert channel, i.e., in the order of 200 bit/s.

4.5. Discussion

In this section, we discuss the limitations of our approach, potential countermeasures, as well as future work.

4.5.1. Limitations

The WebAssembly implementation of port contention offers a lower spatial resolution than the native PortSmash attack proposed by Aldaya et al. [ABuH+19]. Most of this performance loss originates from the challenges introduced by the JavaScript sandbox. In particular, **C3** is the most challenging aspect. Although auxiliary timers offer a very high resolution, they are still inaccurate compared to native cycle-accurate timers. This difference particularly impacts the attack's spatial resolution, as timer imprecision prevents us from measuring small time differences.

Another limitation, inherent to port contention and SMT attacks, is that this attack cannot run in a cross-core setting. We can effectively circumvent **C2** by creating more threads to share a core with the victim, but the attack still depends on the OS scheduler. If the attacker cannot run code on the victim's physical core, the attack does not succeed.

4.5.2. Countermeasures

Many countermeasures have been proposed to mitigate microarchitectural attacks based on SMT. In particular, countermeasures to port contention can effectively apply to web-based port contention. These countermeasures, described more thoroughly in Section 2.6, can stem from several different levels. On the hardware level, SMT could be disabled or propose a more secure way of partitioning resources [TP19, TRVT22]. On the system level, standard approaches, such as port-independent code or static analysis, have been proposed [ABuH+19]. More original approaches, such as an SMT-attacks-aware scheduler, could prevent this side channel. Web-based port contention could also be prevented at the browser level. However, except for timing-based countermeasures (see Chapter 3), browser countermeasures focus on cache side channels or Spectre-like attacks, and have no effect on web port contention.

4.6. Conclusion

We presented the first implementation of port contention in the browser (**Q2**). We showed that port contention side channels have a performance on par or better than previous microarchitectural side channels in the browser, and a more generic threat model. We demonstrated the genericity of this attack by building several types of exploits, including a 200 bit/s covert channel, as well as a concrete example illustrating a side-channel attack with a spatial resolution of 1024 instructions (**Q3**). We further demonstrated the portability of web-based port contention by testing instructions on different Intel CPUs, and we showed that our attack also works in cross-browser and Host-to-VM settings, while being more resilient to noise than cache attacks. We consider port contention side channels, and hardware contention side channels in general, to be a generic class of attacks that can be used as a building block for future microarchitectural attacks in the browser. This work illustrates the difficulty of isolating the JavaScript sandbox from microarchitectural attacks, as currently deployed countermeasures fail to mitigate contention-based side channels.

5 Port Contention Without SMT and its Privacy Implications

Port contention, as described by Aldaya et al. [ABuH+19] and in Chapter 4, requires SMT. Both the attacker and the victim need to run on the same physical core for the attack to work, as CPU ports are on-core resources. If the attacker cannot fill the victim's CPU ports, it cannot detect timing differences caused by the victim's actions, thus cannot mount attacks. This is also valid for the covert channel presented in Section 4.4: if both the sender and receiver cannot use the CPU ports at the same time, they cannot use port contention as the physical layer of the covert channel.

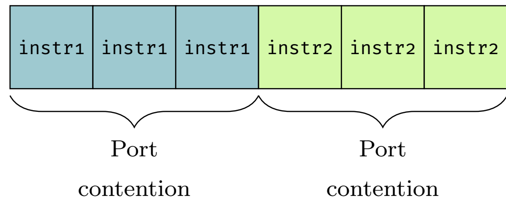
This prerequisite represents a challenge in some settings, as some systems do not have SMT or disable it [Hat], and it may become increasingly hard to fulfill as countermeasures to SMT attacks are being explored [TP19, TRVT22]. It also has severe implications in a web setting, where the attacker script, situated inside the JavaScript sandbox, cannot know nor control which core it is running on. Although we introduced core-control heuristics in Chapter 4, they may not function with a different scheduler, or an SMT-attack aware scheduler.

Contributions This chapter makes the following contributions:

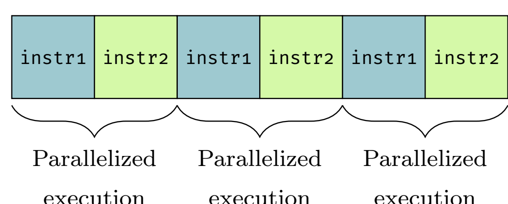
- We present *sequential port contention*, a novel form of contention on the CPU execution ports. It relies on instruction-level parallelism instead of thread-level parallelism, hence does not rely on SMT. We show that such sequences work in native code but also work reliably in WebAssembly. Our side channel works in unmodified off-the-shelf browsers, including the privacy-focused browsers Tor and Brave (Section 5.2).

- We build a framework to automatically find WebAssembly instructions creating sequential port contention. We use differences in port contention behavior to fingerprint the CPU generations in WebAssembly in web browsers without any browser API. We evaluate our new fingerprinting method on 50 CPUs from 12 generations with an accuracy of 95 % with a runtime of only 12 s. We show that our fingerprint is highly stable over major releases of browsers and is robust against system noise (Section 5.3).

- We discuss the security and privacy implications of our new side channel. In particular, we discuss the advantages and limitations of our side channel for fingerprinting applications. We show that sequential port contention is also possible in a virtualized environment but stops working in emulated environments. These results also indicate that the side channel is valuable for malware as an anti-emulation measure (Section 5.4).



(a) *Grouped.* Instructions are executed in batch, creating port contention and reducing parallelism. This results in a slower execution time.



(b) *Interleaved.* Instructions are executed alternatively, allowing them to be executed at the same time, resulting in a faster execution time.

Figure 5.1. – Illustration of the differences in execution time based on the order of instructions, with a look-ahead window of size 1.

5.1. Threat Model

Sequential port contention, as most microarchitectural attacks, requires code execution on the victim machine. We assume that the attacker either has native unprivileged code execution (native side channel) or can run WebAssembly in the victim's browser (browser-based side channel). The attacker does not rely on software vulnerabilities, does not require any permissions that have to be granted by the victim, or any particular setup such as SMT or a specific core assignment. We assume that the victim spends at least 15 s on the attacker's website, based on the average time of 20 s users spend on an unknown website [LWD10].

5.2. Port Contention Without SMT

In this section, we show port contention without requiring SMT, both in a native setting and in a browser sandbox.

5.2.1. Main Idea

The main idea of *sequential port contention* is to exploit the limited look-ahead window of the μop scheduler, leading to contention for well-chosen instruction pairs (`instr1`, `instr2`). Both instructions use different execution ports on the CPU. If the instructions are grouped, i.e., if the instruction stream consists of n instructions `instr1`, followed by n instructions `instr2`, with n larger than the look-ahead window of the scheduler, parallelization is not possible (cf. Figure 5.1a). The scheduler cannot detect that some instructions later in the instruction stream could already be executed in parallel. However, if interleaved in an instruction stream of $2n$ instructions, they can be executed in parallel (cf. Figure 5.1b). As a result, the overall execution time of an instruction stream of the same length depends on the order of the two repeated instructions `instr1` and `instr2` if these instructions do not use the same ports.

Similar to port contention with SMT [ABuH+19], the contending instructions `instr1` and `instr2` depend on the underlying microarchitecture. However, as this information is publicly available [AR19], sequential port contention is applicable to a wide range of microarchitectures. We show sequential port contention in native environments (Section 5.2.2) and demonstrate that it is also exploitable from off-the-shelf unmodified browsers (Section 5.2.3).

Listing 5.1 – *Grouped.* Always creates contention.

```
1  grouped:
2    lfence
3    rdtsc # Timestamp
4    lfence
5    .rept $n # First loop
6      instr1 %reg, %reg
7    .endr
8    .rept $n # Second loop
9      instr2 %reg, %reg
10   .endr
11   lfence # Timestamp
12   rdtsc
```

Listing 5.2 – *Interleaved.* Creates contention if the two instructions share a CPU port.

```
13 interleaved:
14   lfence
15   rdtsc # Timestamp
16   lfence
17
18   .rept $n # Single loop
19     instr1 %reg, %reg
20     instr2 %reg, %reg
21   .endr
22
23   lfence # Timestamp
24   rdtsc
```

5.2.2. Native Environment

5.2.2.1. Proof of Concept

Our proof of concept of sequential port contention is based on two experiments, illustrated in Listings 5.1 and 5.2. In these experiments, we evaluate two native x86 instructions, `instr1` and `instr2`.

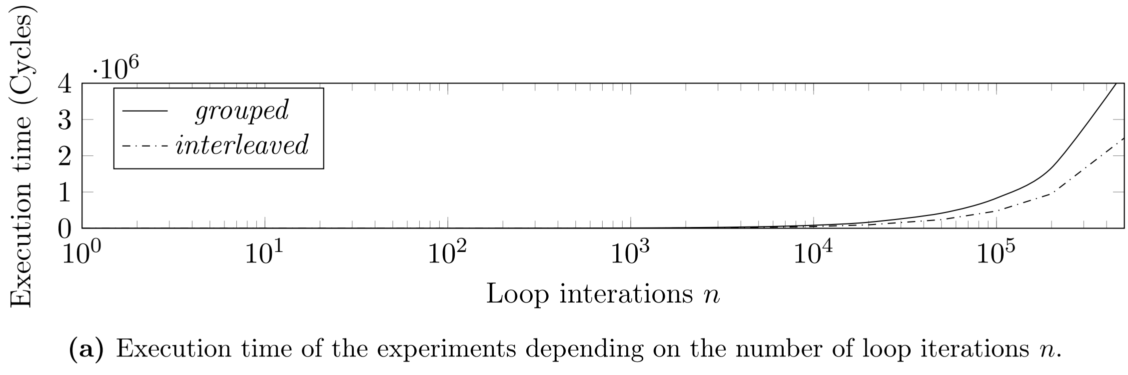
The first experiment is a control experiment, *grouped*, which is composed of two loops, each calling an instruction n times. As the decomposition of instructions in μops is deterministic, the various calls to the same instructions have the same port usage. This means that during loop 1 (respectively loop 2), `instr1` (respectively `instr2`) always creates contention on its ports. The second experiment, *interleaved* is composed of a single loop with the same number of iterations. Instead of calling the same instructions in a row, it alternatively calls `instr1` and `instr2`. If `instr1` and `instr2` emit μops to the same port, it creates contention, resulting in a slower overall execution time. However, if they do not emit μops on the same port, the execution is parallelized due to instruction-level parallelization, resulting in a faster execution time.

By computing $\rho = \frac{time(grouped)}{time(interleaved)}$, we know if *interleaved* creates contention. If $\rho \approx 1$, both experiments have a similar execution time, i.e., the instructions share at least one port. If $\rho > 1$, *interleaved* has a shorter execution time than *grouped*, i.e., the instructions do not share a common port.

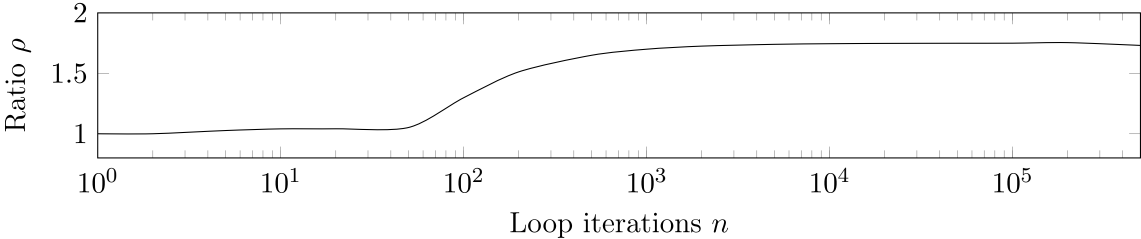
5.2.2.2. Experiments

We ran this experiment on an Intel i5-8365U (Whiskey Lake), with TurboBoost enabled and without fixing the CPU frequency. First, we run it with `instr1` = `crc32`, which emits a single μop on execution port 1 (P1), and `instr2` = `aesdec`, which emits a single μop on execution port 0 (P0). Both instructions have the same throughput and latency.

Figure 5.2 illustrates the results of this experiment when we vary the number of loop iterations n. Figure 5.2a shows how the *grouped* execution time is systematically higher than the *interleaved* one. The gap between the two curves increases with the number of loops. Figure 5.2b shows that ρ quickly converges to 1.8 at $n = 1000$. It then remains constant when increasing the number of loop iterations. The inflection point situated around $n = 64$ is caused by the size of the look-ahead window of the scheduler. When instructions from



(a) Execution time of the experiments depending on the number of loop iterations n.



(b) Ratio ρ depending on the number of loop iterations n.

Figure 5.2. – Sequential port contention experiments for instructions (`crc32`, `aesdec`).

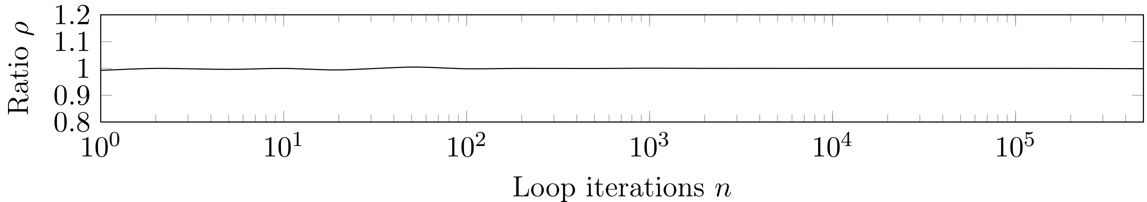


Figure 5.3. – Ratio ρ for (`crc32`, `popcnt`) depending on the loop iterations n.

both loops fit inside this window, the scheduler can rearrange instructions to execute them in the most optimized order, prioritizing parallelism and thus reducing port contention. When an `mfence` is added between the two loops (Lines 7-8 of Listing 5.1), this inflection point disappears, and the curve rises smoothly to 1.8.

We run the same experiment with `instr1` = `crc32` and `instr2` = `popcnt`. Both instructions emit a single P1 μop, and have the same throughput and latency. Figure 5.3 shows that ρ stays constant around 1. That is expected, as the contention is always the same on P1, independently of instruction order.

5.2.3. Web Browsers

5.2.3.1. Challenges

Porting these experiments to a browser sandbox introduces challenges similar to those met in Chapter 4 First, neither JavaScript nor WebAssembly provides high-resolution timers. In this chapter, we use a `SharedArrayBuffer`-based clock. Second, both JavaScript and WebAssembly are high-level languages running inside a sandbox. Moreover, WebAssembly instructions are

Listing 5.3 – *Grouped* in WebAssembly. Always creates contention.

```
25 (module
26  (func $grouped
27  (param $p type)(result type)
28   (local.get $p)
29         (type.instr_1)
30         ... # Repeat $n
31         (type.instr_1)
32         (type.instr_2)
33         ... # Repeat $n
34         (type.instr_2)
35  )
36  (export "grouped" (func $grouped))
37 )
```

Listing 5.4 – *Interleaved* in WebAssembly. Creates contention if the two instructions share a CPU port.

```
38 (module
39  (func $interleaved
40  (param $p type)(result type)
41   (local.get $p)
42         (type.instr_1)
43         ... # Repeat $n
44         (type.instr_1)
45         (type.instr_2)
46         ... # Repeat $n
47         (type.instr_2)
48  )
49  (export "interleaved" (func $interleaved))
50 )
```

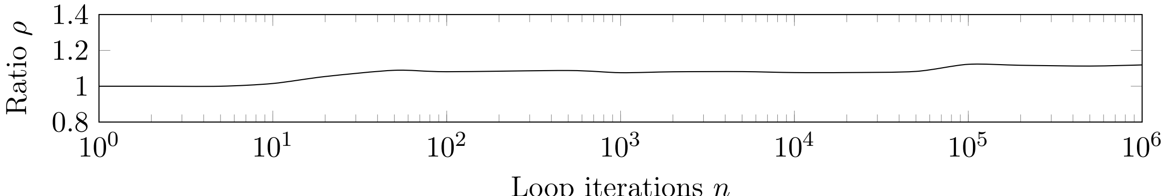


Figure 5.4. – Ratio ρ for the WebAssembly instructions (`i64.popcnt`, `i64.or`) depending on the number of loop iterations n.

an abstraction of native instructions and thus do not directly map to execution ports. As WebAssembly is aimed at being portable, the translation of WebAssembly to native code depends on the browser's WebAssembly compiler and the targeted CPU. We can, however, empirically determine the port usage of these instructions for a system with PC-detector presented in Section 4.2.

5.2.3.2. Proof of Concept

Similar to native experiments, the sequential port contention in WebAssembly is composed of two different functions. Listing 5.3 shows the code for the *grouped* experiment, which results in a slow execution time as instructions are delayed by contention. Listing 5.4 shows the *interleaved* experiment. A low execution time indicates that the experiments were not slowed down by contention, whereas a high execution time means both instructions share at least one port.

5.2.3.3. Experiments

We run this experiment on the same Intel i5-8365U CPU, with `instr1` = `i64.popcnt` and `instr2` = `i64.or`. Figure 5.4 illustrates how ρ also increases with the number of loops. On both Chrome 101 and Firefox 99, ρ stabilizes around 1.1 starting from $n = 100\,000$ loop

iterations. This ratio is significantly lower than the native one. This stems from lower precision timers, as well as the stack structure of WebAssembly, where we need to add a value to the stack between instructions. Running the same experiment with `instr1` = `i64.popcnt` and `instr2` = `i64.ctz`, ρ remains constant around 1 when varying the number of loops. We devise a framework to isolate pairs of instructions that exhibit sequential contention in Section 5.3.2.

Privacy-oriented browsers are also vulnerable to sequential port contention. With 100 000 loop iterations in Brave 1.38, we obtain $\rho = 1.1$. In Tor Browser 11.0.11, `SharedArrayBuffer` is disabled by default to prevent timing attacks. However, we can still reproduce sequential port contention with the lower-resolution timer `performance.now()` by increasing the number of loop iterations n to 100 000 000. In that case, we obtain $\rho = 1.2$, but each experiment takes up to 1 s, i.e., 1000 times more than for other browsers.

5.3. Fingerprinting CPU Generations

In this section, we show how sequential port contention can be used to determine the CPU generation of the victim, even from inside the JavaScript sandbox.

5.3.1. Core Idea

The port usage of native instructions varies across generations of microarchitecture. As the number of execution units and CPU ports vary, the same instruction can emit P1 μops on a given generation and P0 on another generation. For instance, `VPBROADCASTD` emits one μop on P5 on both Haswell and Whiskey Lake microarchitectures, and `AESDEC` emits one μop on P1 on Haswell and one μop on P5 on Whiskey Lake. We computed ρ on an Intel i5-8365U (Whiskey Lake) and an Intel i3-4160T (Haswell). The frequency of these CPUs can vary. However, the base frequency does not impact our experiment as we compute a ratio. We found $\rho_{\text{WhiskeyLake}} = 1$ and $\rho_{\text{Haswell}} = 1.8$. This correlates with the documented port usage. Indeed, on Whiskey Lake, both instructions emit a μop on P5. Thus, both experiments are slowed down by contention. On Haswell, the two instructions do not share a common port. Thus, the *interleaved* experiment is not slowed down by port contention, resulting in a faster execution time and a ratio $\rho > 1$.

In summary, by finding pairs of instructions that create contention for some generations but not others, we can detect on which CPU generation the code is executed. As sequential contention is visible from a browser (cf. Section 5.2.3), we also aim to discover pairs of WebAssembly instructions that exhibit sequential port contention to fingerprint the CPU generation from a web page.

5.3.2. Framework

The port usage of the CPU-independent WebAssembly instructions cannot be determined from the WebAssembly source code. Thus, we build a framework based on PC-detector (see Section 4.2) to automatically evaluate 458 pairs of WebAssembly instructions for contention on a specific CPU generation. Due to the nature of WebAssembly, it is highly portable and can be executed on any microarchitecture. This framework aims at isolating instruction pairs that can act as distinguishers. Such distinguishers have two major properties: 1) they exhibit different contention for different generations, and 2) they always exhibit the same contention for different CPUs of the same generation. The second property is essential, as other sources

of contention that do not depend on the generation could yield false results. Changes in the microarchitecture, e.g., floating-point units, inside a CPU generation can cause changes in behaviors, thus preventing stable fingerprinting.

Using this framework, we collect the best distinguishers to create traces for each generation. To fingerprint generations, we create a k-NN-based classifier and train it with results from the framework. It represents traces as points in an l-dimensional space, where l is the length of the trace, i.e., the number of distinguishers. Given a distance for each unknown execution trace, the classifier computes the k-nearest traces from our training dataset. A trace is classified in the most frequent class, i.e., CPU generation, in the k-nearest-neighbors.

To collect evaluation traces for the two sequential port contention experiments, we use a simple web page (`https://fp-cpu-gen.github.io/fp-cpu-gen`). It works on the latest versions of Firefox and Chrome, on Linux, macOS and Windows.

5.3.3. Evaluation

This section presents the results of our classifier and the different parameters used. Our classifier presents a 95 % accuracy in a real-world threat model (cf. Section 5.1): a user visits a malicious website for a few seconds.

The *training set* is composed of 26 different CPUs spanning 13 different generations. It is composed of both AMD and Intel CPUs, including server and standard desktop CPUs. Table A.1 in Appendix A.3 presents the training set. The *test set* is composed of a subset of traces from the training set. It contains 13 different CPUs. The *evaluation set* is composed of traces from our website. These traces come from an uncontrolled environment since the web script cannot control or quantify the system noise. It contains 50 CPUs from 12 different generations.

5.3.3.1. Training and Testing

We train our model using data from our training set. The CPUs used in our training set are not balanced in terms of CPU generations, some being more represented than others. We therefore include the same number of traces for each generation to compensate for this. Our framework finds 36 pairs of instructions acting as distinguishers between the CPU generations. We use the traces from these distinguishers to train our k-NN classifier. Our model shows a 96 % accuracy on the test set, using $k = 5$ neighbors and majority voting.

5.3.3.2. Accuracy

Figure 5.5 illustrates the results of our classifier on the evaluation set. It has a balanced accuracy of 95 %. We use $k = 5$ as the number of neighbors. We gathered 10 traces and classified each one independently. The class for the experiment is determined using majority voting on the 10 classified traces. Due to the lack of microarchitectural changes between closely-related generations, some generations have the same assignment of execution ports for all instructions. As a consequence, some generations are indistinguishable using sequential port contention. We grouped such generations in the classes of our classifier. This includes the Bridge (Ivy Bridge, Sandy Bridge), Well (Haswell, Broadwell), Skylake (Skylake, Cascade Lake), and Coffee-Lake group (Coffee Lake, Whiskey Lake, Comet Lake). AMD CPUs are distinguishable from Intel ones, but the generations are grouped in the Zen group (Zen 1, Zen 2, and Zen 3).

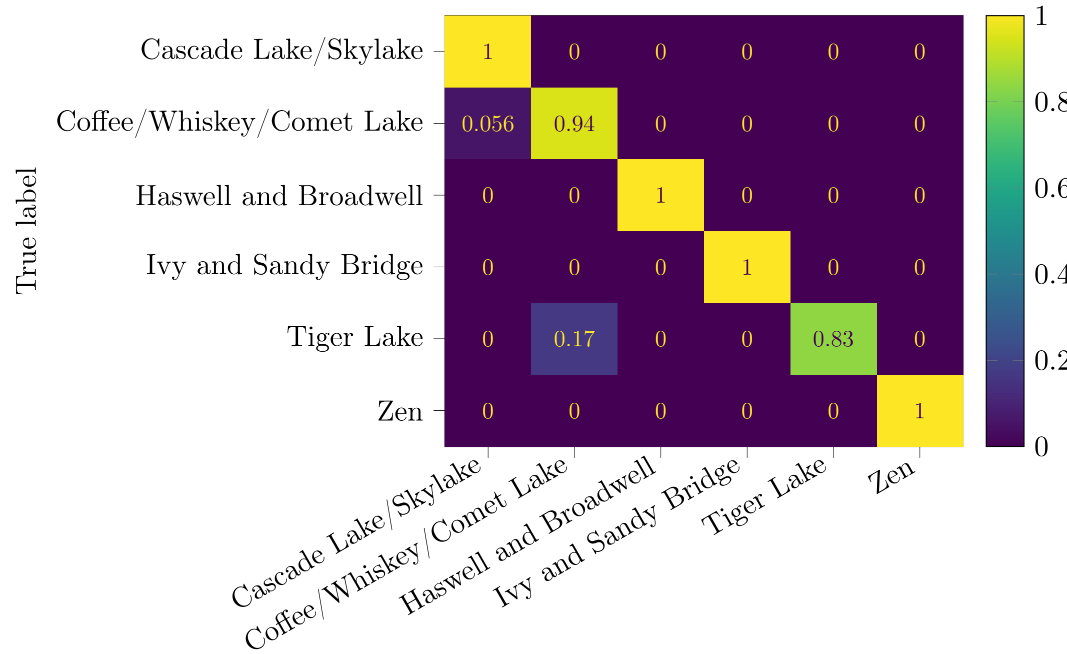


Figure 5.5. – Confusion matrix for the evaluation of the k-NN classifier with grouped generations, $k = 5$ and majority voting on 10 traces.

5.3.3.3. Execution Time

The total execution time is composed of the offline and the online execution time. The *one-time offline execution time* is composed of the framework execution time and k-NN training time. On average, testing all pairs of instructions in the framework takes 4 h, with a standard deviation of 1 hour and 7 minutes. Training the k-NN model takes 5 s on an i5-8365U. The *online execution time* is composed of the data collection time, i.e., the time taken on the website to gather the 10 traces, plus the prediction time, i.e., the execution time of predicting the class of the trace. The data collection time is the most critical factor, as it represents the duration the victim has to remain on the web page. The client sends the traces to the server that then computes the prediction, so the victim can then close the web page. Data collection takes 12 s on average, with a standard deviation of 6 s. The data collection time is faster on average on Google Chrome (10 s) compared to Firefox (13 s). On both browsers, the data collection time is in the range of the average visit time on a website. The prediction time is on average of 40 ms for 10 traces on an i5-8365U, which is negligible compared to the data collection time.

5.3.3.4. Impact of Majority Voting

System noise can decrease the accuracy of a single trace. In particular, the first traces gathered when launching the script are the most prone to these misclassifications. On our evaluation set, the first trace for each experiment has a 30 % chance of being mispredicted, the second 12 %, and then the misclassification rate goes down with repetitions to 6 %. There

are multiple reasons, including the power saving policy of the system, where by default, the CPU does not use its maximum frequency to save energy, and cold caches. The first traces act as a "warm-up" of the CPU, before reaching maximum frequency. To compensate for this phenomenon, we implement majority voting. With majority voting, we gather data on v traces and classify the experiment based on the most common classification of these traces. This improves accuracy at the cost of a higher execution time. Without majority voting, our evaluation set shows an accuracy of 70 % with a data collection time of 5 s. With $v = 5$, the accuracy increases to 86 % and a data collection time of 9 s on average. Starting from $v = 10$, the accuracy peaks at 95 % with a data collection time of 12 s.

5.3.3.5. Impact of the Number of Neighbors

The number of neighbors k is a significant factor in the classifier's efficiency. A small number renders the classification vulnerable to noise, as a single noisy trace in the training set can lead to mispredicting many evaluation traces. A higher number tends to increase the impact of densely grouped traces, as well as increase the computation costs.

For instance, when using $k = 1$ on the evaluation dataset, the classifier accuracy is reduced to 85 %. We found that $k = 5$ grants a higher accuracy for our testing and evaluation sets. Higher values of k tend to yield a lower accuracy. This comes from the similarity of traces between closely-related generation groups, e.g., Skylake and Coffee-Lake groups.

5.3.3.6. Time Stability

Time stability is an essential feature, as hardware is seldomly changed by users, compared to software that has regular updates. We evaluate the stability of the classifier on an Intel i5-8365U on each major version of Firefox and Chrome, covering about a year. The generation is correctly classified from Chrome 91 (released in May 2021) to 101 and Firefox 89 (released in June 2021) to 100. Prior versions did not support WebAssembly SIMD instructions, which are part of the distinguishers in our traces. Our CPU-generation fingerprints have been stable for a year. This represents a high time stability for a browser fingerprint compared to ever-changing browser APIs and other hardware-based approaches, such as DrawnApart [LMD+22], where the fingerprint may change with browsers' major releases, resulting in a median tracking time of 28 days.

5.3.3.7. Impact of Noise on Classification

As the attacker resides inside a sandbox, they cannot know nor control noise created by other processes or tabs. Such noise could deteriorate the performance of our classifier by creating wrong results in the data collection process. We run the data collection process in the website on Firefox 100 and Chrome 101 on a quadcore i5-8365U, while artificially creating noise with the `stress` command. The stress threads create noise, disturbing either the sequential port contention or the clock thread. Fewer noise sources, i.e., `stress -c {1..4}`, result in 93 % accuracy. That is because the OS's scheduler balances the workload, and the attack physical core is not affected by the noise. A higher count of stress threads, i.e., 5 to 8, still yields an accuracy of 75 %.

5.4. Discussion

In this section, we discuss the practical use of CPU generation fingerprinting (Section 5.4.1), its limitations (Section 5.4.2), the effects of virtualization and emulation (Section 5.4.3) as well as possible mitigations (Section 5.4.4).

5.4.1. Practical Use of CPU-Generation Fingerprinting

The CPU-generation attribute does not have a high uniqueness, as even with a bigger training set, there are a limited number of CPU generations. The relevant feature here is its *stability*. We envision using this new fingerprinting attribute in combination with existing attributes. Its stability can be used as a linking factor to better link fingerprints to enhance tracking time [VLRR18] or use fingerprinting as a second authentication factor [LABN19]. Hardware-based fingerprinting attributes are ideal candidates, as hardware is updated less often than software, and software updates usually lead to changes in fingerprints. However, even robust hardware-based methods can break with browser internal changes [LMD+22]. We have shown that our method is robust to major version changes of browsers over a year.

5.4.2. Limitations

For CPU generations with major changes, sequential port contention is a highly reliable method to fingerprint the CPU generation. Such changes are found on Intel CPUs between the Bridge (e.g., Sandy Bridge, Ivy Bridge), the Well (e.g., Haswell, Broadwell), and the Lake (e.g., Skylake, Coffee Lake, Whiskey Lake, Comet Lake) microarchitectures. However, starting with the Lake microarchitecture, changes between new versions are smaller, making it harder to detect the specific microarchitecture. For example, Coffee Lake, Whiskey Lake, and Comet Lake are based on the nearly identical designs of the execution units. Only Ice Lake introduced changes again, specifically with an additional store unit [Wikc] which subsequently led to changes in the port assignment for several instructions. Hence, the detection of the CPU generation cannot differentiate names for essentially the same generation.

Due to lack of access, some generations are not included in the training set, e.g., Nehalem or Ice Lake. Thus, they cannot be correctly predicted by our proof-of-concept model and are not included in the evaluation set. This could be easily corrected by extending our study and running the framework on a larger range of CPUs. CPUs with significant microarchitectural changes are potentially highly identifiable, e.g., Ice Lake with its addition of new store units.

5.4.3. Virtualization and Emulation

Sequential port contention is not limited to bare-metal code execution but also works from inside virtual machines if the guest is virtualized and not emulated.

5.4.3.1. Virtualization

As all involved instructions are unprivileged and not emulated by the hypervisor, there is no difference in the execution stream to a bare-metal execution. Hence, the measured effects are also the same. Moreover, as only a single CPU core is required, the scheduler of the hypervisor does not affect the contention. We verify on Ubuntu 20.04 (kernel 5.13) with QEMU KVM

4.2.1 that we measure the same effect of sequential port contention within a virtual machine (Ubuntu 20.04, kernel 5.4).

5.4.3.2. Emulation

Sequential port contention requires that the specifically-crafted instruction stream is executed without modifications on the CPU. For emulation, this is not the case if instructions are interpreted or translated just in time with potential additional instructions in the instruction stream. For example, when running the guest operating system (Ubuntu 20.04, kernel 5.4) in QEMU 4.2.1 with full system emulation (TCG), we are unable to measure the effect of sequential port contention. In this setup, the instruction stream with and without contention have the same execution time.

Based on this observation, sequential port contention can detect emulation, e.g., if the code is analyzed via a malware-analysis emulator [Kru14, BBR16]. Hence, sequential port contention provides malware with another trick to detect such environments. As discussed in Section 5.4.4, mitigating sequential port contention is difficult. Likewise, sequential port contention is likely infeasible to emulate, making it difficult to prevent malware from detecting the presence of an emulator.

5.4.4. Mitigation

Sequential port contention does not require any operating-system interface or particular setup, such as SMT (cf. Section 5.1). Hence, this side channel cannot be prevented on the operating-system level but potentially on the browser level. Similar to previous work on microarchitectural attack detection [GMWM16, Pay16b, IES18, SBK+21], we show that this side channel can also be detected using hardware performance counters.

5.4.4.1. Browser Mitigation

Existing browser mitigations against side-channel attacks are only effective against sequential port contention if they block access to timing sources [KS16, MCS+16, SLG18a] or entirely prevent the execution of active content [Gio17, Ray17]. However, while effective, these methods also impact the usability of all websites.

The browser can interleave the generated instruction stream with memory fences, effectively preventing out-of-order execution. Theoretically, to fully mitigate the side channel, a browser has to emit a memory fence after every assembly instruction. However, this leads to unacceptable performance penalties, as it effectively prevents out-of-order execution while additionally adding the overhead of the fence (multiple cycles) after every instruction. A trade-off between the number of inserted fences and signal strength might be feasible, though. We leave an evaluation to future work.

Alternatively, the browser can reorder the instruction stream while keeping its functionality. Such reordering can be part of existing compiler optimizations, such as loop optimizations. Software-diversification approaches have also been shown as mitigation against side-channel attacks [CHB+15, RLT15]. As the code required for sequential port contention requires precise control over the instruction sequence, any diversification likely breaks the side channel. We leave the evaluation of software-diversification methods applied by the browser to future work.

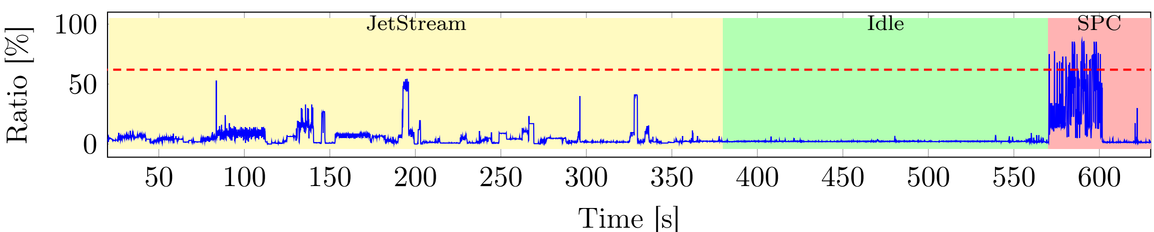


Figure 5.6. – Ratio of backend-bound to misprediction-bound execution when running the JetStream JavaScript and WebAssembly benchmark (left), nothing (middle), and our website for generating the browser fingerprint (right) in Firefox 100.0.2.

5.4.4.2. Detection via Performance Counters

To detect sequential port contention, we propose a metric based on the topdown bottleneck decomposition [Yas14]. Previous work focused mostly on cache-based performance counters for detecting microarchitectural attacks [GMWM16, Pay16b, IES18, SBK+21]. However, for sequential port contention, the cache activity is indistinguishable from typical workloads. The bottleneck exploited in sequential port contention is the execution unit in the backend. As the instruction stream is entirely linear, we use the ratio of backend-bound execution divided by misprediction-bound execution. Hence, the more often the bottleneck is in the backend, combined with next to no mispredictions, the higher the likelihood that the monitored snippet uses sequential port contention.

Figure 5.6 shows the evaluation of this metric in Firefox while running the JetStream JavaScript and WebAssembly benchmark (left), nothing (middle), and our website for generating the browser fingerprint (right). Our tests do not show any workload where this metric is as high as for sequential port contention, allowing detection of this side channel using a simple threshold (dashed line).

5.5. Conclusion

We introduced sequential port contention, a new side channel based on port contention that does not require SMT. This extends the threat surface of this side channel to new systems, and circumvents popular countermeasures. We proposed a WebAssembly framework to automatically determine instruction sequences creating sequential port contention on different systems and CPU vendors. We demonstrated that an attacker can exploit sequential port contention to determine the CPU generation of a victim from the browser within 12 s. This information is highly stable, and the attack works correctly even under heavy system noise. This new side-channel is privacy-threatening, as it is hard to mitigate and can be used for improving the stability of fingerprints. This fingerprint shows how web-based microarchitectural side channels can have an impact on the privacy of the users by reducing the anonymity of the web (**Q3**).

6 Conclusion and Perspectives

This thesis aimed to explore in more depth the threat surface of web-based microarchitectural side channels. To this end, we proposed three major contributions:

1. A systematic study of JavaScript timers and timing attacks (Chapter 3).
2. The implementation of port contention in the browser (Chapter 4).
3. A new evolution of this side channel, and the study of its impact on privacy (Chapter 5).

With these contributions, we aimed to explore the research questions **Q1**, **Q2**, and **Q3**.

Q1: What is the current landscape of browser-based attacks and countermeasures? Browsers are constantly evolving pieces of software, and the security countermeasures change with every release. This makes it particularly hard to evaluate the state of browsers' security to microarchitectural side channels. Chapter 3 was primarily focused on this question, as it proposes a classification of timing attacks, countermeasures, as well as tools to evaluate timers in JavaScript automatically.

Furthermore, we also believe that a single, fixed-in-time study is insufficient to make durable contributions to browser security, as attacker code can be executed on different browsers of different versions, on different OSs sitting on different hardware. To that extent, we provided automated evaluation frameworks for our proofs-of-concepts, in order to make them as portable and durable as possible. PC-detector (Chapter 4) allows detecting the contention caused by WebAssembly instructions regardless of the browser versions (as long as WebAssembly is supported). Our sequential port contention framework (Chapter 5) can evaluate instructions even on non-x86 processors, and use it to automatically detect the generation of the processor.

Q2: Are other components vulnerable to side channels from the JavaScript sandbox? Cache attacks have probably been the literature's most explored microarchitectural side channels. However, new components are exploited from the JavaScript sandbox to widen the threat surface of web-based side channels. Generally, these side channels are first implemented in native code, as it offers a more fine-grained control over the microarchitecture.

This is the case with web-based port contention. Aldaya et al. [ABuH+19] theorized native CPU port contention, and in Chapter 4 we extended it to the JavaScript sandbox. This contribution shows that there are still threatening side channels than could be implemented in JavaScript. In particular, JavaScript port contention shows that SMT side channels can be implemented in JavaScript as an attacker can achieve core co-residency with a victim, even without core control.

Sequential port contention, defined in Chapter 5, was directly designed to run in the JavaScript sandbox. It allows to extend port contention to new systems, where SMT attacks are unavailable. With it, we show how a change in the paradigm of a side channel, here from thread-level parallelism to instruction-level parallelism, can offer a new threat surface to already discovered side channels.

Q3: What can an attacker extract from these side channels? This is probably the most pragmatic of this manuscript's research questions, as it aims to quantify the impact of microarchitectural side channels in browsers. With this thesis, we tried to exemplify how flexible these attacks are, and that they can apply to many different security fields.

Microarchitectural side channels have often been applied to attacks against cryptographic algorithms, even in the JavaScript sandbox [GPTY18]. We developed the threat of web port contention on this type of attack in our artificial example in Section 4.3.

However, we believe that side channels, and in particular port contention, can also have heavy implications on web users' privacy. This is first the case with the port-contention covert channel explored in Section 4.4. This channel completely breaks the isolation barriers brought by the sandbox and site isolation, as it allows an attacker in the sandbox to communicate with the outside. It could be used to extract native information to the sandbox, to leak Tor Browser users' IP addresses, or to exchange authentication cookies between two different tabs. We also explored port contention's impact on another privacy aspect: browser fingerprinting. In Chapter 5, we showed how an attacker could leverage sequential port contention to identify the CPU generation of the victim, allowing to complement software-based fingerprints and grant more extended time stability.

Perspectives and Future Work

This manuscript has provided partial answers to the research questions introduced in the introduction. However, if these contributions have not provided definitive answers to the open questions, they have brought even more perspectives for future work.

Of new side channels and new threats The first lead for future work would be to explore the implementation of new side channels to web browsers. This research is critical as each microarchitectural component can be exploited differently, bringing different threats and ways to circumvent already existing mitigations. Efficient countermeasures cannot be implemented without a clear view of the attack surface.

With the resolution of JavaScript-specific challenges, e.g., lack of high-resolution timers or core-control, the exploitation of new microarchitectural components is more accessible. In particular, on-core resources already exploited natively, e.g., the TLB or μop cache, could be exploited from the browser to widen the attack surface.

Hardware fingerprinting We briefly explored how microarchitectural side channels could be used for fingerprinting. This direction is highly interesting as fingerprinting can be seen as a specific type of side channel, where the attacker collects hardware information instead of secrets.

Hardware fingerprints are currently starting to be explored [LMD+22] as it offers different properties than software attributes. In particular, hardware attributes are more stable, as

users do not change their CPU monthly. This stability permits to link less stable software fingerprints in time. Hardware attributes are also less spoofable because they depend on active fingerprinting: instead of reading an attribute that the user can modify, the information results from actual computations in the browser. Although a user can still modify these values to prevent fingerprinting, it is harder to set the specific fingerprint of another system, thus preventing spoofing.

Hardware fingerprints can be decomposed in two different directions based on the type of attributes. The first direction would be to use precise information about the hardware to complement software fingerprinting. This is the same direction we explored with sequential port contention by finding the generation of the processor. Using microarchitectural side channels, an attacker could detect the CPU or GPU model, vendor, or attributes about microarchitectural properties, e.g., cache size or associativity, or the number of CPU cores. All this information does not create a unique fingerprint as many users have the same processors, but is stable information about a set of discrete attributes. The second direction would be to create more unique fingerprints based on hardware imperfections. This is already the direction explored by Sánchez-Rola et al. [SSB18] and Laor et al. [LMD+22]. By leveraging minor differences in execution time or results of non-deterministic computations, a tracker could gather unique information about a user. This type of fingerprint is often less stable than discrete attributes, as other phenomena, such as software or system noise, can impact the fingerprint.

This new paradigm in fingerprinting broadens the scope of applications for fingerprinting. In particular, its stability and relative resistance to spoofing could help develop fingerprint as a multi-factor authentication token. However, its implications on tracking applications are also heavy, as it eases long-lasting fingerprints, allowing a third party to track anonymous users over the web.

Browser-based countermeasures: a deadend? Web-based microarchitectural attacks stand at a paradoxical crossroads: they are low-level attacks exploited from a very high-level sandbox. Logically, they can be mitigated at several levels. Whereas microarchitectural countermeasures are currently the most studied as they also apply to native attacks, they are hard to implement as it is not possible to patch hardware.

Browser-based countermeasures can be deployed rapidly after the discovery of attacks. However, the current state of browser-based countermeasures is unclear. Timing-based countermeasures were a popular paradigm after the discovery of the first microarchitectural side channels in the browser, but are now abandoned as they are often too penalizing for web developers. The current paradigm for browser-based countermeasures is based on isolation, specifically targetting same address-space attacks, such as Spectre-PHT. As presented in Chapter 3, this new direction is insufficient, as it does not prevent most timing attacks. However, browser vendors seem to consider microarchitectural countermeasures as a dead-end, as implementing hardware-level countermeasures at an application level is often too costly.

A new approach we would like to explore in future work is based on detecting attacks instead of preventing them. In particular, approaches similar to performance-counters-based approaches [CSY16, AYQ+16, PIO19, LG18, For18, WSS+20] could be efficient to protect the web. In a third-party attack, the browser could detect untrusted processes and monitor their behavior to detect suspicious patterns before the attack is over. This approach could be based on reading performance counters in the browser process, or implementing new

policies on suspicious array accesses. Although this type of detection is based on heuristics, machine-learning approaches could provide the genericity needed to detect the wide variety of web-based microarchitectural side channels, and could be trained and deployed rapidly.

Automation and systematization Security research resembles a cat-and-mouse game: new attacks motivate new countermeasures, which in turn force the creation of newer attacks. The field of web-based microarchitectural channels is not an exception and is rapidly evolving.

In particular, each new side channel requires reverse-engineering the behavior of a microarchitectural component, finding a vulnerability, exploiting it natively then modifying it so it can be exploited from the JavaScript sandbox. This workflow requires extensive engineering work, often highly repetitive from one attack to another. We believe that at least some side channels could be detected automatically by an automated framework, comparable to the native ABSynthe [GGK+20] or Osiris [WIN+21]. These frameworks rely on a theoretical systematization of side channels to evaluate random pairs of instructions for data leakage. The same idea could be done on JavaScript and WebAssembly instructions. PC-detector is a preliminary implementation of this idea, but is limited to detecting on-core contention between pairs of instructions. A more complex implementation of a black-box side-channel detector could provide the literature with a more complete picture of sources of leakages in the sandbox. Another future work on automated discovery of side channels would be implementing the detector at the application level. Instead of evaluating WebAssembly or JavaScript instructions, evaluating the output bytecode produced by JavaScript engines could bring a more deterministic understanding of how the sandbox interacts with the microarchitecture.

We believe this systematic approach is necessary to produce effective and generic countermeasures instead of specific patches for a particular type of attack.

Appendices A

A.1. Custom RDTSC implementation

We modified the following files in order to implement `rdtsc`:

A.1.1. Firefox 81

mozilla-central/dom/performance/Performance.cpp

```
std::uint64_t Performance::Rdtsc(){
    unsigned int lo,hi;
    __asm__ __volatile__ ("mfence");
    __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
    __asm__ __volatile__ ("mfence");
    return ((std::uint64_t)hi << 32) | lo;
}
```

mozilla-central/dom/performance/Performance.h

```
std::uint64_t Rdtsc();
```

mozilla-central/dom/webidl/Performance.webidl

```
typedef double uint64_t;
uint64_t rdtsc();
```

A.1.2. Chromium 84

chromium/src/third_party/blink/renderer/core/timing/performance.cc

```
std::uint64_t Performance::rdtsc(){
    unsigned int lo,hi;
    __asm__ __volatile__ ("mfence");
    __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
    __asm__ __volatile__ ("mfence");
    return ((std::uint64_t)hi << 32) | lo;
}
```

chromium/src/third_party/blink/renderer/core/timing/performance.h

```
std::uint64_t rdtsc();
```

chromium/src/third_party/blink/renderer/core/timing/dom_high_res_time_-stamp.idl

```
typedef double uint64_t;
```

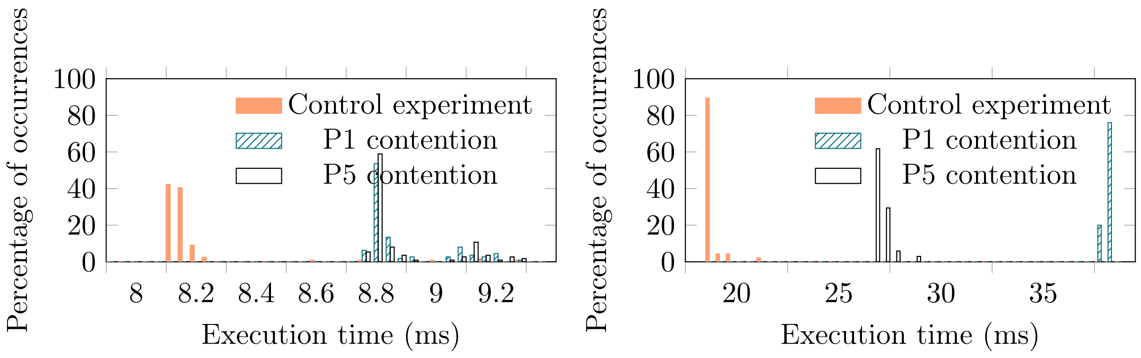
chromium/src/third_party/blink/renderer/core/timing/performance.idl

```
uint64_t rdtsc();
```

A.2. Port Contention on Other WebAssembly Instructions

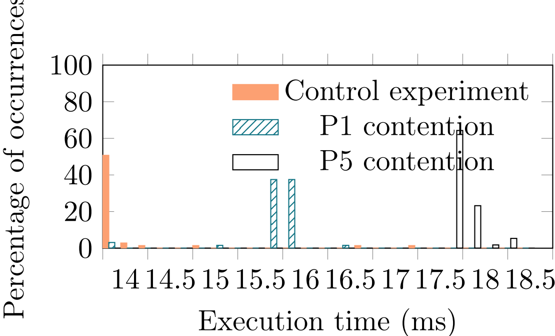
Figures A.1a to A.1c show port contention on the following WebAssembly instructions: `f64.floor`, the pair `f32.convert_i32_u` and `i32.trunc_f32_u`, and `i64.rem_u`. We can clearly distinguish the three outcomes of a PC-detector usage:

- Figure A.1a illustrates an instruction that do not cause contention. The P1 and P5 distributions have a similar mean and standard deviation, making them difficult to distinguish. However, they are still distinguishable from the control experiment.
- Figure A.1c illustrates a pair of instructions causing contention on P5. The distribution P5 has a higher mean than P1 and the control experiment.
- Figure A.1b illustrates an instruction causing contention on P1. The distribution P1 has a higher mean than P5 and the control experiment.



(a) P1 contention experiment on `f64.floor` for 1 000 000 instructions.

(b) P1 contention experiment on `i64.rem_u` for 1 000 000 instructions.



(c) P5 contention experiment on paired `f32.convert_i32_u` and `i32.trunc_f32_u` for 1 000 000 instructions.

A.3. Training set

Table A.1 presents the different processors used to train the model presented in Section 5.3.2.

Table A.1. – CPUs used in our training set

CPU	Vendor	Generation
Xeon X5670	Intel	Westmere
Xeon E5-2620	Intel	Sandy Bridge
Xeon E5-2630	Intel	Sandy Bridge
Xeon E5-2630L	Intel	Sandy Bridge
Xeon E5-2650 0	Intel	Sandy Bridge
Xeon E5-2660 0	Intel	Sandy Bridge
Core i5-2520M	Intel	Sandy Bridge
Xeon E5-2660 v2	Intel	Ivy Bridge
Xeon E5-2630 v3	Intel	Haswell
Core i3-4160T	Intel	Haswell
Xeon E5-2620 v4	Intel	Broadwell
Xeon E5-2630 v4	Intel	Broadwell
Xeon E5-2680 v4	Intel	Broadwell
Core i3-5010U	Intel	Broadwell
Xeon Gold 6126	Intel	Skylake
Xeon Gold 6130	Intel	Skylake
Core i9-9980HK	Intel	Coffee Lake
Core i5-8365U	Intel	Whiskey Lake
Xeon Gold 5218	Intel	Cascade Lake SP
Xeon Gold 5220	Intel	Cascade Lake SP
Core i7-10510U	Intel	Comet Lake
Core i7-10710U	Intel	Comet Lake
Core i5-1135G7	Intel	Tiger Lake
EPYC 7301	AMD	Zen
Ryzen 5 2500U	AMD	Zen
Ryzen 9 5900HX	AMD	Zen 3

Bibliography A

[AAG17] Zirak Allaf, Mo Adda, and Alexander Gegov. A comparison study on flush+ reload and prime+ probe attacks on aes using machine learning approaches. In *UK Workshop on Computational Intelligence*, pages 203–213. Springer, 2017.

[ABG10] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 110–124. Springer, 2010.

[ABuH+19] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *S&P*, 2019.

[AEE+14] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juárez, Arvind Narayanan, and Claudia Díaz. The web never forgets: Persistent tracking mechanisms in the wild. In *CCS*, 2014.

[AES15] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. S$a: A shared cache attack that works across cores and defies VM sandboxing - and its application to AES. In *IEEE Symposium on Security and Privacy*, pages 591–604. IEEE Computer Society, 2015.

[AIES15] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In *AsiaCCS*, pages 85–96. ACM, 2015.

[AJ] Anne van Kesteren Artur Janc, Charlie Reis. Coop and coep explained. https://docs.google.com/document/d/1zDlfvfTJ_9e8Jdc8ehuV4zMEu9ySMCiTGMS9y0GU92k/edit. Accessed: 2022-10-05.

[AKM+15] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *S&P*, 2015.

[AKS07] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *CT-RSA*, volume 4377 of *Lecture Notes in Computer Science*, pages 225–242. Springer, 2007.

[ANT+20] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. Ladderleak: Breaking ECDSA with less than one bit of nonce leakage. In *CCS*, pages 225–242. ACM, 2020.

[AR] Andreas Abel and Jan Reineke. Tzcnt uops.info page. https://uops.info/html-instr/TZCNT_R16_R16.html. Accessed: 2021-11-11.

[AR19] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS*, pages 673–686. ACM, 2019.

[AYQ+16] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd M. Austin. ANVIL: software-based protection against next-generation rowhammer attacks. In *ASPLOS*, pages 743–755. ACM, 2016.

[BB05] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 2005.

[BBR16] Michael Brengel, Michael Backes, and Christian Rossow. Detecting hardware-assisted virtualization. In *DIMVA*, volume 9721 of *Lecture Notes in Computer Science*, pages 207–227. Springer, 2016.

[BEPW10] Andrey Bogdanov, Thomas Eisenbarth, Christof Paar, and Malte Wienecke. Differential cache-collision timing attacks on AES with applications to embedded cpus. In *CT-RSA*, volume 5985 of *Lecture Notes in Computer Science*, pages 235–251. Springer, 2010.

[Ber61] Jay M. Berger. A note on error detection codes for asymmetric channels. *Inf. Control.*, 4(1):68–73, 1961.

[Ber05] Daniel Bernstein. Cache-timing attacks on aes. 2005.

[BFS20] Daniel De Almeida Braga, Pierre-Alain Fouque, and Mohamed Sabt. Dragonblood is still leaking: Practical cache-based side-channel in the wild. In *ACSAC*, pages 291–303. ACM, 2020.

[BFS21] Daniel De Almeida Braga, Pierre-Alain Fouque, and Mohamed Sabt. PARASITE: password recovery attack against srp implementations in the wild. In *CCS*, pages 2497–2512. ACM, 2021.

[BH09] Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *ASIACRYPT*, volume 5912 of *Lecture Notes in Computer Science*, pages 667–684. Springer, 2009.

[BM18] Sarani Bhattacharya and Debdeep Mukhopadhyay. *Advanced Fault Attacks in Software: Exploiting the Rowhammer Bug*, pages 111–135. 2018.

[BMD+17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. *CoRR*, abs/1702.07521, 2017.

[BMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, pages 991–1008. USENIX Association, 2018.

[BRBG16] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *IEEE Symposium on Security and Privacy*, pages 987–1004. IEEE Computer Society, 2016.

[BRPG15] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN: silently breaking ASLR in the cloud. In *WOOT*. USENIX Association, 2015.

[BSN+19] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *CCS*, 2019.

[Bug18a] Bugzilla. Reduce timer resolution to 2ms. https://bugzilla.mozilla.org/show_bug.cgi?id=1435296, feb 2018.

[Bug18b] Bugzilla. Set timer resolution to 1ms with jitter. https://bugzilla.mozilla.org/show_bug.cgi?id=1451790, apr 2018.

[Bug18c] Bugzilla. Unanticipated security/usability degradation from precision-lowering of performance.now() to 2ms. https://bugzilla.mozilla.org/show_bug.cgi?id=1440863, feb 2018.

[Bug20] Bugzilla. Check crossoriginisolated for all nsrfpservice::reducetimeprecision* callers. https://bugzilla.mozilla.org/show_bug.cgi?id=1586761, 02 2020.

[BvdPSY14] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "ooh aah... just a little bit" : A small amount of side channel can go a long way. In *CHES*, volume 8731 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2014.

[CBS+19] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security Symposium*, pages 249–266. USENIX Association, 2019.

[CCLW17] Yinzhi Cao, Zhanhao Chen, Song Li, and Shujiang Wu. Deterministic browser. In *CCS*, pages 163–178. ACM, 2017.

[CEQZ06] Feng Cao, Martin Ester, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *SDM*, pages 328–339. SIAM, 2006.

[CGG+19] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*, 2019.

[CHB+15] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *NDSS*, 2015.

[Chr17a] Chromium. Issue 611420: WebAccessibleResources take too long to make a decision about loading if the extension is installed. https://bugs.chromium.org/p/chromium/issues/detail?id=611420, 2017.

[Chr17b] Chromium. Issue 709464: Detecting the presence of extensions through timing attacks (including Incognito) - Chromium bug tracker. https://bugs.chromium.org/p/chromium/issues/detail?id=709464, 2017.

[Cona] MDN Contributors. Cross-origin-embedder-policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Embedder-Policy. Accessed: 2021-19-11.

[Conb] MDN Contributors. Cross-origin-opener-policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy. Accessed: 2021-19-11.

[Conc] MDN Contributors. Cross-origin resource policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Cross-Origin_Resource_Policy_(CORP).

[Cond] MDN Contributors. Date.now() api. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/now. Accessed: 2022-06-30.

[Cone] MDN Contributors. Navigator.hardwareconcurrency. https://developer.mozilla.org/en-US/docs/Web/API/Navigator/hardwareConcurrency. Accessed: 2021-19-11.

[Conf] MDN Contributors. Planned changes to shared memory. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer/Planned_changes. Accessed: 2022-07-26.

[Cong] MDN Contributors. Same-origin-policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. Accessed: 2020-10-06.

[Conh] MDN Contributors. Subresource integrity. https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity.

[Coni] MDN Contributors. Window.requestanimationframe() api. https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame. Accessed: 2022-06-30.

[Conj] MDN Contributors. Window.settimeout() api. https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout. Accessed: 2022-06-30.

[Con20a] MDN Contributors. Performance api. https://developer.mozilla.org/en-US/docs/Web/API/Performance_API, October 2020.

[Con20b] MDN Contributors. Performance.now() api. https://developer.mozilla.org/fr/docs/Web/API/Performance/now, Sept 2020.

[CSH+20] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: break it, fix it, repeat. In *AsiaCCS*, pages 481–493. ACM, 2020.

[CSY16] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Appl. Soft Comput.*, 2016.

[Dam06] Peter Damaschke. Threshold group testing. In *GTIT-C*, volume 4123 of *Lecture Notes in Computer Science*, pages 707–718. Springer, 2006.

[DFS20] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. Hybcache: Hybrid side-channel-resilient caches for trusted execution environments. In *USENIX Security Symposium*, 2020.

[DJL+12] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Archit. Code Optim.*, 8(4):35:1–35:21, 2012.

[DKPT17] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. Prime+abort: A timer-free high-precision L3 cache attack using intel TSX. In *USENIX Security Symposium*, pages 51–67. USENIX Association, 2017.

[Eck10] Peter Eckersley. How unique is your web browser? In *Privacy Enhancing Technologies*, 2010.

[ECMa] ECMA. Atomics.add - standard. https://www.ecma-international.org/ecma-262/#sec-atomics.add. Accessed: 2020-09-30.

[ECMb] ECMA. Sharedarraybuffer objects. https://tc39.es/ecma262/#sec-sharedarraybuffer-objects. Accessed: 2022-06-30.

[ECMc] ECMA. Standard ecma-262. https://tc39.es/ecma262/. Accessed: 2022-07-21.

[ECMd] ECMA. Standard ecma-262. https://tc39.es/ecma402/. Accessed: 2022-07-21.

[EP16] Dmitry Evtyushkin and Dmitry V. Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *CCS*, pages 843–857. ACM, 2016.

[EPA16] Dmitry Evtyushkin, Dmitry V. Ponomarev, and Nael B. Abu-Ghazaleh. Jump over ASLR: attacking branch predictors to bypass ASLR. In *MICRO*, pages 40:1–40:13. IEEE Computer Society, 2016.

[ERAP18] Dmitry Evtyushkin, Ryan Riley, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *ASPLOS*, pages 693–707. ACM, 2018.

[F+11] Agner Fog et al. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. *Copenhagen University College of Engineering*, 93:110, 2011.

[FE15] David Fifield and Serge Egelman. Fingerprinting web users through font metrics. In *Financial Cryptography*, volume 8975 of *Lecture Notes in Computer Science*, pages 107–124. Springer, 2015.

[Fog] Andreas Fogh. Covert shotgun. https://cyber.wtf/2016/09/27/covert-shotgun/. Accessed: 2022-07-04.

[For18] James Christopher Foreman. A survey of cyber security countermeasures using hardware performance counters. *CoRR*, abs/1807.10868, 2018.

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

[GBM15] Daniel Gruss, David Bidner, and Stefan Mangard. Practical memory deduplication attacks in sandboxed javascript. In *ESORICS (1)*, volume 9326 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2015.

[GGK+20] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020.

[GIA+15] Berk Gülmezoglu, Mehmet Sinan Inci, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. A faster and more realistic flush+reload attack on AES. In *COSADE*, volume 9064 of *Lecture Notes in Computer Science*, pages 111–126. Springer, 2015.

[Gio17] Giorgio Maone. NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience!, July 2017.

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *DIMVA*, volume 9721 of *Lecture Notes in Computer Science*, pages 300–321. Springer, 2016.

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *DIMVA*, volume 9721 of *Lecture Notes in Computer Science*, pages 279–299. Springer, 2016.

[Gooa] Google. Google end to end library. https://github.com/google/end-to-end/. Accessed: 2022-07-26.

[Goob] Google. Product status: Microarchitectural data sampling (mds). https://support.google.com/faqs/answer/9330250?hl=en. Accessed: 2021-19-11.

[Gooc] Google. Sharedarraybuffer updates in android chrome 88 and desktop chrome 92 - chrome developers. https://developer.chrome.com/blog/enabling-shared-array-buffer/. Accessed: 2022-07-26.

[Good] Google. time_clamper.cpp. https://source.chromium.org/chromium/chromium/src/+/master:third_party/blink/renderer/core/timing/time_clamper.cc. Accessed: 2020-10-12.

[Gooe] Google. V8 javascript engine. https://v8.dev/. Accessed: 2022-07-21.

[Goof] Google. v8/bytecodes.h. https://github.com/v8/v8/blob/master/src/interpreter/bytecodes.h. Accessed: 2022-07-26.

[GPTY18] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *ACNS*, 2018.

[GRB+17] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*. The Internet Society, 2017.

[GRBG18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX*, 2018.

[Gri] Ilya Grigorik. High resolution time level 2. https://www.w3.org/TR/hr-time-2/. Accessed: 2022-06-30.

[Gro] W3C Community Group. Index of instructions webassembly 2.0. https://webassembly.github.io/spec/core/appendix/index-instructions.html. Accessed: 2022-05-20.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*, pages 897–912. USENIX Association, 2015.

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO (1)*, volume 8616 of *Lecture Notes in Computer Science*, pages 444–461. Springer, 2014.

[GZ13] Michael Misiu Godfrey and Mohammad Zulkernine. A server-side solution to cache-based side-channel attacks in the cloud. In *IEEE CLOUD*, pages 163–170. IEEE Computer Society, 2013.

[Ham50] Richard W Hamming. Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160, 1950.

[Hat] Red Hat. Disabling smt to prevent cpu security issues using the web console. https://access.redhat.com/documentation/en-us/red-hat-enterprise-linux/8/topic/f1d65124-781b-4543-a51a-d2bf9fa794ac. Accessed: 2022-05-10.

[Hor] Jan Horn. Project zero: Reading privileged memory with a side channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html. Accessed: 2022-07-04.

[Hor18] Jann Horn. speculative execution, variant 4: speculative store bypass, 2018.

[HS13] Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. In *CARDIS*, volume 8419 of *Lecture Notes in Computer Science*, pages 219–235. Springer, 2013.

[Hu92] Wei-Ming Hu. Lattice scheduling and covert channels. In *IEEE Symposium on Security and Privacy*, pages 52–61. IEEE Computer Society, 1992.

[HWH13] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy*, pages 191–205. IEEE Computer Society, 2013.

[IES16] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *AsiaCCS*, pages 353–364. ACM, 2016.

[IES18] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: preventing microarchitectural attacks before distribution. In *CODASPY*, pages 377–388. ACM, 2018.

[IIES14] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-vm attack on aes. In *International Workshop on Recent Advances in Intrusion Detection*, pages 299–319. Springer, 2014.

[Inta] Intel. Affected processors: Transient execution attacks & related security. https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html. Accessed: 2022-07-26.

[Intb] Intel. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures white paper - ia-32-ia-64-benchmark-code-execution-paper.pdf. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf. Accessed: 2022-07-26.

[Intc] Intel. Intel® 64 and ia-32 architectures optimization reference manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Accessed: 2022-06-30.

[Intd] Intel. Intel® 64 and ia-32 architectures software developer's manual. https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf. Accessed: 2022-07-20.

[Inte] Intel. Introduction to cache allocation technology. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html. Accessed: 2022-07-26.

[JS12] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *IEEE Symposium on Security and Privacy*, pages 143–157. IEEE Computer Society, 2012.

[K15] Sakamoto K. Reduce resolution of performance.now to prevent timing attacks. https://bugs.chromium.org/p/chromium/issues/detail?id=506723, Jul 2015.

[KASZ09] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *HPCA*, 2009.

[KDK+14] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, pages 361–372. IEEE Computer Society, 2014.

[KGA+20] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Netcat: Practical cache attacks from the network. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 20–38. IEEE, 2020.

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

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[KKSA18] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *WOOT @ USENIX Security Symposium*. USENIX Association, 2018.

[Koc96] Paul Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[KPM12] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *USENIX Security Symposium*, pages 189–204. USENIX Association, 2012.

[Kru14] Christopher Kruegel. Full system emulation: Achieving successful automated dynamic analysis of evasive malware. In *BlackHat USA*, 2014.

[KS16] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Security Symposium*, pages 463–480. USENIX Association, 2016.

[KSWH00] John Kelsey, Bruce Schneier, David A. Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. *J. Comput. Secur.*, 8(2/3):141–158, 2000.

[KW18] Vladimir Kiriansky and Carl A. Waldspurger. Speculative buffer overflows: Attacks and defenses. *CoRR*, abs/1807.03757, 2018.

[Kyö18] Sami Kyöstilä. Clamp performance.now() to 100us. `https://chromium-review.googlesource.com/c/chromium/src/+/849993`, Jan 2018.

[LABN19] Pierre Laperdrix, Gildas Avoine, Benoit Baudry, and Nick Nikiforakis. Morellian analysis for browsers: Making web authentication stronger with canvas fingerprinting. In *DIMVA*, 2019.

[Lar] Michael Larabel. Openbsd disabling smt / hyper threading due to security concerns. `https://www.phoronix.com/scan.php?page=news_item&px=OpenBSD-Disabling-SMT`. Accessed: 2021-19-11.

[LAS+18] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing rowhammer faults through network requests. *CoRR*, abs/1805.04956, 2018.

[LG18] Congmiao Li and Jean-Luc Gaudiot. Online detection of spectre attacks using microarchitectural traces from performance counters. In *SBAC-PAD*, pages 25–28. IEEE, 2018.

[LGR14] Peng Li, Debin Gao, and Michael K. Reiter. Stopwatch: A cloud architecture for timing channel mitigation. *ACM Trans. Inf. Syst. Secur.*, 17(2):8:1–8:28, 2014.

[LGS+16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *USENIX Security Symposium*, pages 549–564. USENIX Association, 2016.

[LGS+17] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed javascript. In *ESORICS (2)*, volume 10493 of *Lecture Notes in Computer Science*, pages 191–209. Springer, 2017.

[LHS+20] Moritz Lipp, Vedad Hadzic, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take A way: Exploring the security implications of amd's cache way predictors. In *AsiaCCS*, pages 813–825. ACM, 2020.

[LKO+21] Moritz Lipp, Andreas Kogler, David F. Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: software-based power side-channel attacks on x86. In *IEEE Symposium on Security and Privacy*, pages 355–371. IEEE, 2021.

[LL14] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *MICRO*, 2014.

[LMD+22] Tomer Laor, Naif Mehanna, Antonin Durey, Vitaly Dyadyuk, Pierre Laperdrix, Clémentine Maurice, Yossi Oren, Romain Rouvoy, Walter Rudametkin, and Yuval Yarom. DRAWNAPART: A device identification technique based on remote GPU fingerprinting. In *NDSS*, 2022.

[LSG+20] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. Meltdown: reading kernel memory from user space. *Commun. ACM*, 63(6):46–56, 2020.

[LWD10] Chao Liu, Ryen W. White, and Susan T. Dumais. Understanding web browsing behaviors through weibull analysis of dwell time. In *SIGIR*, 2010.

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622. IEEE Computer Society, 2015.

[May09] Jonathan R Mayer. Any person... a pamphleteer": Internet anonymity in the age of web 2.0. *Undergraduate Senior Thesis, Princeton University*, 85, 2009.

[MBYS11] Keaton Mowery, Dillon Bogenreif, Scott Yilek, and Hovav Shacham. Fingerprinting information in JavaScript implementations. In Helen Wang, editor, *Proceedings of W2SP 2011*. IEEE Computer Society, May 2011.

[MCS+16] Jian Mao, Yue Chen, Futian Shi, Yaoqi Jia, and Zhenkai Liang. Toward exposing timing-based probing attacks in web applications. In *WASA*, volume 9798 of *Lecture Notes in Computer Science*, pages 499–510. Springer, 2016.

[MOG+20] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel SGX. In *IEEE Symposium on Security and Privacy*, pages 1466–1482. IEEE, 2020.

[Moza] Mozilla. Rabaldrmonkey baseline compile. https://hg.mozilla.org/mozilla-central/file/tip/js/src/wasm/WasmBaselineCompile.cpp. Accessed: 2022-07-26.

[Mozb] Mozilla. Spidermonkey javascript engine. https://spidermofnkey.dev/. Accessed: 2022-07-21.

[Mozc] Mozilla. v8/bytecodes.h. https://searchfox.org/mozilla-central/source/js/src/vm/Opcodes.h. Accessed: 2022-07-26.

[Moz19] Mozilla. Always restyle / repaint when a visited query finishes – Mozilla Central. https://hg.mozilla.org/mozilla-central/rev/89fad029456188f03a670ef5f08a5d0856a728b1, 2019.

[Moz20a] Mozilla. Bug 884270: Link Visitedness can be detected by redraw timing – Bugzilla. https://bugzilla.mozilla.org/show_bug.cgi?id=884270, 2020.

[Moz20b] Mozilla. nsrfpservice.cpp, firefox sourcecode. https://hg.mozilla.org/mozilla-central/file/tip/toolkit/components/resistfingerprinting/nsRFPService.cpp, October 2020.

[MR18] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *CCS*, pages 2109–2122. ACM, 2018.

[MWS+17] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: SSH over robust cache covert channels in the cloud. In *NDSS*, 2017.

[NS06] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *Selected Areas in Cryptography*, volume 4356 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2006.

[NSY15] Gabi Nakibly, Gilad Shelef, and Shiran Yudilevich. Hardware fingerprinting using HTML5. *CoRR*, abs/1503.01408, 2015.

[OACD15] Lukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Díaz. The leaking battery - A privacy analysis of the HTML5 battery status API. In *DPM/QASA@ESORICS*, volume 9481 of *Lecture Notes in Computer Science*, pages 254–263. Springer, 2015.

[OKSK15] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *CCS*, pages 1406–1418. ACM, 2015.

[Ope] OpenPGP.js. Openpgp.js | openpgp javascript implementation. https://openpgpjs.org/. Accessed: 2022-07-26.

[Ore15] Yossi Oren. "spy in the sandbox" - security issue related to high resolution time api. https://bugzilla.mozilla.org/show_bug.cgi?id=1167489, may 2015.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *CT-RSA*, pages 1–20. Springer, 2006.

[OW11] Rodney Owens and Weichao Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *IPCCC*, pages 1–8. IEEE Computer Society, 2011.

[Pag05] Dan Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR Cryptol. ePrint Arch.*, page 280, 2005.

[Pay16a] Mathias Payer. Hexpads: A platform to detect "stealth" attacks. In *ESSoS*, volume 9639 of *Lecture Notes in Computer Science*, pages 138–154. Springer, 2016.

[Pay16b] Mathias Payer. Hexpads: A platform to detect "stealth" attacks. In *ESSoS*, volume 9639 of *Lecture Notes in Computer Science*, pages 138–154. Springer, 2016.

[Per05] Colin Percival. Cache missing for fun and profit. In *In Proc. of BSDCan 2005*, 2005.

[PGGV21] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *S&P*, 2021.

[PGM+16] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: exploiting DRAM addressing for cross-cpu attacks. In *USENIX Security Symposium*, pages 565–581. USENIX Association, 2016.

[PIO19] Iván Prada, Francisco D Igual, and Katzalin Olcoz. Detecting time-fragmented cache attacks against aes using performance monitoring counters. In *Conference on Cloud Computing and Big Data*, pages 3–15. Springer, 2019.

[PLF21] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. In *USENIX Security Symposium*, pages 645–662. USENIX Association, 2021.

[PSS+18] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. In *EuroS&P*, pages 338–352. IEEE, 2018.

[PTV21] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+scope: Overcoming the observer effect for high-precision cache contention attacks. In *CCS*, pages 2906–2920. ACM, 2021.

[Qur18] Moinuddin K. Qureshi. CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping. In *MICRO*, pages 775–787. IEEE Computer Society, 2018.

[Ray17] Raymond Hill. uBlock Origin - An efficient blocker for Chromium and Firefox. Fast and lean., July 2017.

[RGB+16] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security Symposium*, pages 1–18. USENIX Association, 2016.

[RLT15] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital {Side-Channels} through obfuscated execution. In *USENIX Security Symposium*, 2015.

[RMBO22] Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. Port contention goes portable: Port contention side channels in web browsers. In *AsiaCCS*, pages 1182–1194. ACM, 2022.

[RML21] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. Sok: In search of lost time: A review of javascript timers in browsers. In *EuroS&P*, pages 472–486. IEEE, 2021.

[RMO19] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. In *USENIX Security Symposium*, 2019.

[RMS22] Thomas Rokicki, Clémentine Maurice, and Michael Schwarz. Cpu port contention without SMT. In *ESORICS*, 2022.

[RMT+21] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. I see dead μops: Leaking secrets via intel/amd micro-op caches. In *ISCA*, 2021.

[RR01] Josyula R. Rao and Pankaj Rohatgi. Empowering side-channel attacks. *IACR Cryptol. ePrint Arch.*, page 37, 2001.

[RRR16] Michael Rushanan, David Russell, and Aviel D. Rubin. Malloryworker: Stealthy computation and covert channels using web workers. In *STM*, volume 9871 of *Lecture Notes in Computer Science*, pages 196–211. Springer, 2016.

[SAO+21] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+probe 1, javascript 0: Overcoming browser-based side-channel defenses. In *USENIX Security Symposium*, 2021.

[SBK+21] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Daniel Gruss, and Michael Schwarz. Dynamic process isolation. *CoRR*, abs/2110.04751, 2021.

[SD15] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 15:71, 2015.

[Sea] M. Seaborn. How physical addresses map to rows and banks in dram. http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html. Accessed: 2022-07-04.

[sel20] SeleniumHQ browser automation. https://www.selenium.dev/, oct 2020.

[SIYA11] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory deduplication as a threat to the guest OS. In *EUROSEC*, page 1. ACM, 2011.

[SKH+19] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security Symposium*, 2019.

[SKLG21] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. Remote memory-deduplication attacks. *NDSS*, 2021.

[SLCO18] Bin Shi, Bo Li, Lei Cui, and Liu Ouyang. Vanguard: A cache-level sensitive file integrity monitoring system in virtual machine environment. *IEEE Access*, 6:38567–38577, 2018.

[SLG18a] Michael Schwarz, Moritz Lipp, and Daniel Gruss. Javascript zero: Real javascript and zero side-channel attacks. In *NDSS*. The Internet Society, 2018.

[SLG+18b] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. Keydrown: Eliminating software-based keystroke timing side-channel attacks. In *NDSS*. The Internet Society, 2018.

[SLG19] Michael Schwarz, Florian Lackner, and Daniel Gruss. Javascript template attacks: Automatically inferring host information for targeted exploits. In *NDSS*, 2019.

[SLKN19] Oleksii Starov, Pierre Laperdrix, Alexandros Kapravelos, and Nick Nikiforakis. Unnecessarily identifiable: Quantifying the fingerprintability of browser extensions due to bloat. In *WWW*, 2019.

[SLM+19] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *CCS*, pages 753–768. ACM, 2019.

[SMAK20] Benjamin Semal, Konstantinos Markantonakis, Raja Naeem Akram, and Jan Kalbantner. Leaky controller: Cross-vm memory controller covert channel on multi-core systems. In *SEC*, volume 580 of *IFIP Advances in Information and Communication Technology*, pages 3–16. Springer, 2020.

[SMGM17] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *Financial Cryptography*, volume 10322 of *Lecture Notes in Computer Science*, pages 247–267. Springer, 2017.

[SNK+12] Alexander Schlösser, Dmitry Nedospasov, Juliane Krämer, Susanna Orlic, and Jean-Pierre Seifert. Simple photonic emission analysis of AES - photonic side channel analysis for the rest of us. In *CHES*, volume 7428 of *Lecture Notes in Computer Science*, pages 41–57. Springer, 2012.

[SP13] Raphael Spreitzer and Thomas Plos. Cache-access pattern attack on disaligned AES t-tables. In *COSADE*, volume 7864 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2013.

[SP18] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking FPU register state using microarchitectural side-channels. *CoRR*, abs/1806.07480, 2018.

[SQ21] Gururaj Saileshwar and Moinuddin K. Qureshi. MIRAGE: mitigating conflict-based cache attacks with a practical fully-associative design. In *USENIX Security Symposium*, 2021.

[SS20] Milad Seddigh and Hadi Soleimany. Enhanced flush+reload attack on AES. *ISC Int. J. Inf. Secur.*, 12(2):81–89, 2020.

[SSB17a] Iskander Sánchez-Rola, Igor Santos, and Davide Balzarotti. Extension breakdown: Security analysis of browsers extension resources control policies. In *USENIX Security Symposium*, pages 679–694. USENIX Association, 2017.

[SSB17b] Iskander Sánchez-Rola, Igor Santos, and Davide Balzarotti. Extension breakdown: Security analysis of browsers extension resources control policies. In *USENIX Security Symposium*, pages 679–694. USENIX Association, 2017.

[SSB17c] Iskander Sánchez-Rola, Igor Santos, and Davide Balzarotti. Extension breakdown: Security analysis of browsers extension resources control policies. In *USENIX Security Symposium*, 2017.

[SSB18] Iskander Sánchez-Rola, Igor Santos, and Davide Balzarotti. Clock around the clock: Time-based device fingerprinting. In *CCS*, 2018.

[SSL+19] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*, pages 279–299. Springer, 2019.

[ST04] Adi Shamir and Eran Tromer. Acoustic cryptanalysis, 2004.

[Sta] Chrome Platform Status. Align performance api timer resolution to cross-origin isolated capability. https://chromestatus.com/feature/6497206758539264.

[Sto13] Paul Stone. Pixel perfect timing attacks with HTML5, 2013.

[SVS17] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. Discovering browser extensions via web accessible resources. In *CODASPY*, pages 329–336. ACM, 2017.

[SWG+17] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. *CoRR*, abs/1702.08719, 2017.

[SWT01] Dawn Xiaodong Song, David A. Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security Symposium*. USENIX, 2001.

[Teaa] ARM Security Team. Speculative processor vulnerability. https://developer.arm.com/Arm%20Security%20Center/Speculative%20Processor%20Vulnerability. Accessed: 2022-07-04.

[Teab] Google Security Team. Leaky page. https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html. Accessed: 2022-07-04.

[THAC18] David Trilla, Carles Hernández, Jaume Abella, and Francisco J. Cazorla. Cache side-channel attacks and time-predictability in high-performance critical real-time systems. In *DAC*, pages 98:1–98:6. ACM, 2018.

[TKA+18] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *USENIX Annual Technical Conference*, pages 213–226. USENIX Association, 2018.

[TP19] Daniel Townley and Dmitry Ponomarev. SMT-COP: defeating side-channel attacks on execution units in SMT processors. In *PACT*, 2019.

[TRVT22] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. Secsmt: Securing SMT processors against contention-based covert channels. In *USENIX Security Symposium*, 2022.

[TS] Ben L. Titzer and Jaroslav Sevcik. A year with spectre: a v8 perspective. https://v8.dev/blog/spectre. Accessed: 2021-02-12.

[TSS17] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. CLKSCREW: exposing the perils of security-oblivious energy management. In *USENIX Security Symposium*, pages 1057–1074. USENIX Association, 2017.

[Tur18] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. *URL https://support.google.com/faqs/answer/7625886*, 2018.

[Vah] Lutz Vahl. Intent to extend origin trial: Trial for sharedarraybuffers in non-isolated pages on desktop platforms. https://mikewest.github.io/cookie-incrementalism/draft-west-cookie-incrementalism.htmlhttps://groups.google.com/a/chromium.org/g/blink-dev/c/OLbI-axDyH0/m/qYYz7LvEAgAJ. Accessed: 2022-08-05.

[vba] v8 blog. Ignition - v8. https://v8.dev/docs/ignition. Accessed: 2022-07-26.

[vbb] v8 blog. Liftoff - v8. https://v8.dev/blog/liftoff. Accessed: 2022-07-26.

[vbc] v8 blog. Turbofan - v8. https://v8.dev/docs/turbofan. Accessed: 2022-07-26.

[vdPSY15] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In *CT-RSA*, volume 9048 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2015.

[VDS11] Bhanu Chandra Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in xen. In *CCSW*, pages 41–46. ACM, 2011.

[vGJ17] Tom van Goethem and Wouter Joosen. One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions. In *WOOT*. USENIX Association, 2017.

[vGJN15] Tom van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *CCS*, pages 1382–1393. ACM, 2015.

[VK17] Pepe Vila and Boris Köpf. Loophole: Timing attacks on shared event loops in chrome. In *USENIX Security Symposium*, 2017.

[VKM19] Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. In *IEEE Symposium on Security and Privacy*, pages 39–54. IEEE, 2019.

[VLRR18] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. FP-STALKER: tracking browser fingerprint evolutions. In *S&P*, 2018.

[VRS14] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael M. Swift. Scheduler-based defenses against cross-vm side-channels. In *USENIX Security Symposium*, 2014.

[vSMÖ+19] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue in-flight data load. In *S&P*, 2019.

[W3C] W3C. Webassembly core specification. https://webassembly.github.io/spec/core/bikeshed/. Accessed: 2022-07-26.

[w3t] w3techs. Usage statistics of javascript as client-side programming language on websites. https://w3techs.com/technologies/details/cp-javascript/. Accessed: 2022-07-26.

[Wag18] Luke Wagner. Mitigations landing for new class of timing attack. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/, jan 2018.

[WC14] Ruisheng Wang and Lizhong Chen. Futility scaling: High-associativity cache partitioning. In *MICRO*, pages 356–367. IEEE Computer Society, 2014.

[Wes] Mike West. Incrementally better cookies. https://mikewest.github.io/cookie-incrementalism/draft-west-cookie-incrementalism.html. Accessed: 2020-11-05.

[WFZ+16] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Secdcp: secure dynamic cache partitioning for efficient timing channel protection. In *DAC*, pages 74:1–74:6. ACM, 2016.

[Wika] Mozilla Wiki. https://www.mozilla.org/en-US/firefox/95.0/releasenotes/.

[Wikb] Perf Wiki. Perf. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2022-07-26.

[Wikc] WikiChip. Sunny cove - microarchitectures - intel - wikichip. https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove. Accessed: 2022-05-20.

[WIN+21] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated discovery of microarchitectural side channels. In *USENIX Security Symposium*, pages 1415–1432. USENIX Association, 2021.

[WL07] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*, pages 494–505. ACM, 2007.

[WSS+20] Han Wang, Hossein Sayadi, Avesta Sasan, Setareh Rafatirad, Tinoosh Mohsenin, and Houman Homayoun. Comprehensive evaluation of machine learning countermeasures for detecting microarchitectural side-channel attacks. In *ACM Great Lakes Symposium on VLSI*, pages 181–186. ACM, 2020.

[WZCN20] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. Dramdig: A knowledge-assisted tool to uncover DRAM address mapping. In *DAC*, pages 1–6. IEEE, 2020.

[XXHW13] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. Security implications of memory deduplication in a virtualized environment. In *DSN*, pages 1–12. IEEE Computer Society, 2013.

[XZZT16] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *USENIX Security Symposium*, pages 19–35. USENIX Association, 2016.

[Yas14] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *ISPASS*, pages 35–44. IEEE Computer Society, 2014.

[YB14] Yuval Yarom and Naomi Benger. Recovering openssl ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. *IACR Cryptol. ePrint Arch.*, page 140, 2014.

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

[ZJOR11] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *IEEE Symposium on Security and Privacy*, pages 313–328. IEEE Computer Society, 2011.

[ZR13] Yinqian Zhang and Michael K. Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS*, 2013.

[ZW09] Kehuan Zhang and XiaoFeng Wang. Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *USENIX Security Symposium*, pages 17–32. USENIX Association, 2009.

Titre : Canaux Auxiliaires dans les Navigateurs : Applications à la Sécurité et la Vie Privée

Mot clés : Canaux auxiliaires, Microarchitecture, Sécurité du Web, Contention de port, Attaques par minutage

Résumé :
Les attaques par canal auxiliaire exploitent les effets secondaires d'un calcul sensible pour divulguer des secrets. Leur implémentation dans les navigateurs web représente une augmentation considérable de la surface d'attaque, mais s'accompagne de défis dus à l'environnement restrictif et aux mises à jour constantes des navigateurs. Cette thèse évalue la menace que représentent les canaux auxiliaires microarchitecturaux dans les navigateurs.

Nous fournissons une systématisation des attaques par minutage et des contre-mesures dans les navigateurs. En particulier, nous montrons que l'évolution de la sécurité basée sur le timing rend les navigateurs plus vulnérables aux attaques par minutage.

Nous présentons également la contention de port du processeur dans l'environnement restreint de JavaScript . Notre canal auxiliaire a une résolution spatiale comparable à celle des meilleurs canaux auxiliaires dans les navigateurs et permet la création d'un canal caché à large bande passante. En outre, nous montrons que la contention de port peut également être exploitée sans SMT. Les implications de ce nouveau canal auxiliaire sur la vie privée sont inquiétantes, car un attaquant peut l'utiliser pour récupérer les générations de processeurs.

Title: Side Channels in Web Browsers: Applications to Security and Privacy

Keywords: Side channels; Microarchitecture, Web Security, Port Contention, Timing attacks

Abstract: Side channel attacks exploit the side effects of sensitive computation to leak secrets. Their implementation in web browsers represents a considerable increase in threat surface, but comes with challenges due to the restrictive environment and the constant browser updates. This thesis evaluates the threat posed by microarchitectural side channels in browsers.

We provide a systematization of timing attacks and countermeasures in browsers. With automatic frameworks, we show that the shift in timing-based security makes browsers more vulnerable to timing attacks.

We also introduce CPU port contention to the JavaScript Sandbox. Our side channel has a spatial resolution on par with the best side channels in the browsers and allows the creation of a high-bandwidth covert channel. Furthermore, we show that port contention can also be leveraged without SMT. The implications of this new side channel on privacy are worrying, as an attacker can leverage it to retrieve CPU generations.