THÈSE

en vue de l'obtention du grade de

Doctorat en Sciences Informatiques
Université de Lille & Inria & CRIStAL

Discipline : **Informatique**

Laboratoire CRIStAL

École Doctorale MADIS

Présentée et soutenue publiquement le 26 novembre 2025
par **Maxime Huyghe**

Exploration automatique de l'impact des paramètres de configuration sur les empreintes de navigateur

---

Directeur de Thèse : Lionel Seinturier
Co-directeur de Thèse : Clément Quinton
Co-encadrant : Walter Rudametkin

Devant la commission d'examen formée de :

M. Lionel Seinturier	*Professeur à l'Université de Lille*	Directeur
M. Clément Quinton	*Maître de conférence HDR à l'Université de Lille*	Co-directeur
M. Walter Rudametkin	*Professeur à l'Université de Rennes*	Co-encadrant
Mme Sonia Ben Mokhtar	*Directrice de recherche LIRIS*	Présidente
M. Aurélien Francillon	*Professeur à Eurecom*	Rapporteur
M. Vincent Roca	*Chargé de recherche Inria*	Rapporteur
Mme Estelle Pawlowski	*Maître de conférence HDR à l'ENSICAEN*	Examinatrice

Laboratoire CRIStAL UMR 9189
Université de Lille, Cité Scientifique,
Batiment Esprit,
59655 Villeneuve-d'Ascq

École Graduée (doctoral) MADIS
Université de Lille, Cité Scientifique,
Bâtiment. P3,
59655 Villeneuve d'Ascq

Remerciements

Depuis longtemps, je souhaitais atteindre cet objectif, et je l'ai aujourd'hui atteint.

Je tiens à remercier mes encadrants :

Walter pour ses conseils, son partage et pour m'avoir encouragé à aller plus loin (même si j'ai commencé à les appliquer pour les crédits ECTS, et explosé le quota), et d'avoir cru en moi depuis le début. Clément pour son soutien et son accompagnement (même pendant les longues soirées avant les deadlines), ainsi que sa capacité à me rassurer (nous avons réussi à trouver la sortie de la jungle). Lionel pour son efficacité et son accompagnement, pour m'avoir intégré à l'équipe Spirals (je ne sais toujours pas comment tu fais pour être aussi rapide à répondre aux mails et aux messages).

Je remercie également les membres du jury :

Vincent et Aurélien pour avoir lu en détail mon manuscrit et pour avoir enrichi nos échanges. Estelle et Sonia pour leurs questions pertinentes, et nos échanges sur différents aspects de mes travaux.

Sans oublier tous les membres de l'équipe Spirals :

Naif, pour m'avoir parrainé sans le savoir et pour avoir éclairci le chemin. Mes collègues de bureau (et plus) : Hugo, avec qui j'ai longuement parlé de vim, de Linux et du cloud. Brell pour nos débats (à fond dans l'IA) et le partage d'expériences en photographie. Sihem, toujours présente (encore merci pour le pot de la soutenance) et souriante. Iliana et Tristan pour nos longues heures de débat (scientifique ou non), qui nous remettent en question et nous poussent à voir plus loin. Alexandre pour ses conseils en mécanique (et sa capacité à transformer les minutes en heures). Rémy et Nathan pour les soirées et leur joie de vivre. Adrien pour ses échanges politiques, associatifs et techniques. Pierre pour ses conseils sur les postes après la thèse, et pour le fingerprinting.

J'ai sûrement oublié des personnes, mais sachez que chaque personne que j'ai rencontrée m'a permis d'avancer, de me remettre en question et, ce faisant, d'atteindre mes objectifs. Merci à tous pour avoir contribué à faire de ma thèse une merveilleuse expérience.

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Web browsers expose numerous API attributes, forming unique fingerprints that enable user tracking without consent. As traditional methods like cookies face restrictions, browser fingerprinting has become a persistent alternative leveraging browser configurations and execution environments. Yet, the impact of configuration parameters on fingerprint characteristics is poorly understood, hindering privacy-enhancing technology development and informed user decisions.

This thesis addresses these challenges using empirical analysis and formal modeling to advance our understanding of browser fingerprinting and configuration variability. We present three main contributions offering comprehensive insights into browser configurations and fingerprint characteristics.

1. First, FP-Rainbow, an automated methodology for analyzing how browser configuration parameters affect fingerprint characteristics. Through extensive experimentation across $1,748$ configuration parameters and 18 versions of Chromium browser, we generated $61,559$ browser fingerprints representing the most comprehensive systematic study of browser configuration impact to date. Our analysis identified 32 to 56 impactful parameters on the Browser Object Model and achieved 78.15% success in identifying configuration parameters from fingerprints, quantitatively proving configuration choices directly impact fingerprints.

2. Second, we present a novel formal representation using feature modeling from Software Product Line Engineering to capture browser fingerprint variability. Our method transforms flat attribute-value pairs into hierarchical trees, preserving relationships and constraints. This approach enables advanced analysis of browser fingerprint variability, like fingerprint sampling, evolution tracking, and configuration space exploration, it also has a side effect of reducing the size by up to $59,000$ times without loss of information.

3. Third, we propose extending these feature modeling techniques to configuration parameters and environmental attributes (OS, Hardware, Software). By unifying all data, we introduce a query language and a method to extract minimal fingerprints that maintain high identification rates.

These contributions have implications across multiple domains. For privacy, our findings offer quantitative evidence for informed user decisions on browser configu-

rations. Browser developers can use our tools to assess new features privacy impact and implement privacy-preserving default configurations.

All datasets, tools, and methodologies are open-source to ensure reproducible research and community validation. The comprehensive datasets include detailed hardware and software configuration information alongside browser fingerprints, facilitating diverse research applications. This commitment fosters a broader understanding of browser fingerprinting and supports developing effective privacy-enhancing technologies.

This work opens promising future directions: extending to other browsers/platforms, longitudinal studies of fingerprint evolution, integrating machine learning for improved analysis, and developing next-generation privacy-enhancing technologies. Our methodologies provide a foundation for ongoing privacy assessment as web technologies evolve.

Résumé

Les navigateurs web exposent de nombreux attributs d'API, créant des empreintes numériques uniques pour le suivi non consenti des utilisateurs. Face aux restrictions des méthodes traditionnelles (ex. cookies), les empreintes de navigateur s'imposent comme une alternative persistante, exploitant configurations et environnements d'exécution. L'impact des paramètres de configuration sur les empreintes est toutefois mal compris, freinant le développement de technologies de protection de la vie privée et la prise de décisions éclairées par les utilisateurs.

Cette thèse aborde ces défis par une analyse empirique et une modélisation formelle pour approfondir la compréhension des empreintes de navigateur et de sa variabilité. Nous présentons trois contributions majeures, offrant des perspectives sur les configurations et les caractéristiques des empreintes.

1. Premièrement, FP-Rainbow, une méthodologie automatisée analysant l'impact des paramètres de configuration sur les empreintes de navigateur. Une expérimentation sur 1 748 paramètres et 18 versions de Chromium a généré 61 559 empreintes, constituant l'étude systèmatique la plus exhaustive à ce jour sur l'impact des configurations. Notre analyse a identifié 32 à 56 paramètres clés qui impactent l'empreinte et a atteint $78,15\%$ de succès dans l'identification des paramètres à partir des empreintes du navigateur, prouvant leur impact direct et quantifié.

2. Deuxièmement, nous introduisons une nouvelle représentation formelle par modélisation par caractéristiques (feature modeling) issue de l'ingénierie des lignes de produits logiciels pour capturer la variabilité des empreintes numériques. Notre méthode transforme les paires attribut-valeur en arbres hiérarchiques, préservant relations et contraintes. Elle facilite l'analyse avancée de la variabilité (échantillonnage, suivi d'évolution, exploration de l'espace de configuration) et réduit la taille des données jusqu'à 59 000 fois sans perte d'information.

3. Troisièmement, nous étendons ces techniques aux paramètres de configuration et aux attributs environnementaux (OS, matériel, logiciel). En unifiant ces données, nous introduisons un langage de requête et une méthode pour extraire des empreintes minimales conservant des taux d'identification élevés.

Ces contributions ont des implications multidisciplinaires. Pour la confidentialité, nos résultats offrent des preuves quantitatives pour éclairer les décisions des

utilisateurs. Les développeurs peuvent utiliser nos outils pour évaluer l'impact sur la vie privée des nouvelles fonctionnalités et implémenter des configurations par défaut protectrices.

Tous nos jeux de données, outils et méthodologies sont open-source, assurant reproductibilité et validation communautaire. Ces jeux de données complets intègrent informations détaillées sur les configurations matérielles, logicielles et les empreintes, facilitant diverses applications de recherche. Cette démarche favorise une compréhension approfondie des empreintes de navigateurs et soutient le développement de technologies efficaces de protection de la confidentialité.

Ces travaux ouvrent des perspectives prometteuses : extension à d'autres navigateurs/plateformes, études longitudinales de l'évolution des empreintes, intégration du machine learning pour une analyse affinée, et développement de technologies de confidentialité de nouvelle génération. Nos méthodologies posent les bases d'une évaluation continue de la confidentialité face à l'évolution des technologies web.

Chapter 1

Introduction

1.1 Context and Problem Statement

The modern web has evolved into a complex ecosystem where billions of users interact daily with countless online services, applications, and platforms. Over the past two decades, this digital transformation has fundamentally changed how we communicate, work, shop, and access information [4, 5]. However, this evolution has also introduced significant privacy and security challenges that affect the vast majority of users worldwide [6, 7]

1.1.1 Evolution of the Web and Privacy Challenges

Web browsers have transformed from simple document viewers into sophisticated software platforms capable of running complex applications, handling multimedia content, and providing rich interactive experiences [8, 9]. Modern browsers like Chrome, Firefox, Safari, and Edge, which collectively represent over 95% of the global market share,[1] have become one of the primary gateways to the digital world alongside mobile applications. These browsers process enormous volumes of HTTP requests daily with Google Chrome alone handling an estimated several billion requests across its user base while managing substantial amounts of personal data including browsing histories, cookies, and user preferences [10, 11].

This evolution has been accompanied by an increasing demand for personalized services and targeted content delivery [12]. Online businesses rely heavily on user identification and tracking technologies to provide customized experiences, deliver relevant advertisements, and analyze user behavior patterns [13, 14]. However, this reliance on user identification has created a well-documented tension between service personalization and user privacy protection [15, 16].

The privacy landscape has become increasingly complex since the mid-2010s as users have become more aware of data collection practices [17]. According to Eurostat data from 2021, while 80% of European internet users aged 16-74 were aware that cookies can be used to trace online activities, only 36% had actually

[1] https://gs.statcounter.com/browser-market-share

modified their browser settings to prevent or limit cookie usage.[2] This awareness has been further heightened by major data breaches such as the Cambridge Analytica scandal (2018), the Equifax breach (2017), and other privacy incidents, alongside the implementation of comprehensive data protection regulations including the General Data Protection Regulation (GDPR) in 2018 and the California Consumer Privacy Act (CCPA) in 2020 [18, 19].

1.1.2 Limitations of Traditional Tracking Methods

Since the mid-1990s, web tracking has relied primarily on persistent identifiers such as HTTP cookies, which allowed websites to recognize returning visitors and maintain session state across browsing sessions [20, 21]. First-party cookies, set by the domain being directly visited by the user, enabled essential website functionality including user authentication, session management, and shopping cart persistence [22]. In contrast, third-party cookies, set by external domains embedded within web pages (typically advertising networks and analytics providers), enabled cross-site tracking and behavioral profiling for targeted advertising purposes [23, 24].

However, traditional cookie-based tracking faces several limitations:

- **User Control**: Users can easily disable, block, or delete cookies through browser settings or privacy extensions [25, 26].
- **Regulatory Pressure**: Privacy regulations increasingly require explicit consent for non-essential cookies, limiting their effectiveness [27, 28].
- **Browser Policies**: Major browser vendors have implemented varying approaches to third-party cookie restrictions: Safari began blocking third-party cookies by default in 2017 through Intelligent Tracking Prevention (ITP), Firefox implemented Enhanced Tracking Protection in 2019, while Google Chrome initially planned to phase out third-party cookies by 2024 but reversed this decision in July 2024, opting instead to introduce user choice mechanisms.[3][4][5]
- **Technical Limitations**: Cookies are domain-specific and cannot provide cross-browser or cross-device tracking without additional mechanisms [29, 30].
- **Storage Restrictions**: Cookies have size limitations and can be easily cleared by users or browser maintenance procedures [22].

Other traditional tracking methods, such as IP address tracking, face similar challenges. IP addresses change regularly due to dynamic allocation, are shared among multiple users in corporate or residential networks, and can be masked through VPN services or proxy servers [31, 32].

[2] https://ec.europa.eu/eurostat/web/products-eurostat-news/-/edn-20220208-1
[3] https://privacysandbox.com/news/privacy-sandbox-update/
[4] https://blog.google/products/chrome/updated-timeline-privacy-sandbox-milestones/
[5] https://blog.google/products/chrome/building-a-more-private-web/

1.1.3 Emergence of Browser Fingerprinting as an Alternative

In response to the limitations of traditional tracking methods, browser fingerprinting has gained prominence as an alternative tracking technique since the early 2010s [29]. This technique leverages the distinctive characteristics of web browsers and their host systems to create probabilistically unique identifiers without requiring stored data on the user's device [33].

A browser fingerprint 3.2 is a collection of attributes and values pairs that uniquely identify a user's browser and their environment. Browser fingerprinting exploits the fact that each browser instance exposes substantial information about its configuration, capabilities, and environment through standard web APIs[6] and HTTP headers [34]. Much of this information is accessible via the Browser Object Model (BOM), a collection of objects (*e.g.*, `window`, `navigator`, `screen`) that provide programmatic access to browser features and the document that we can access via JavaScript. The BOM (distinct from the DOM, which exclusively represents the `document` object) is organized hierarchically, with each object relating to a parent or child object. The root object is `window`, serving as the global object within the browser's context. Figure 1.1 illustrates this structure.

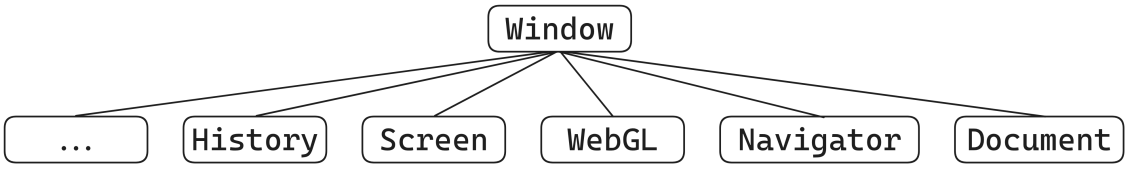


Figure 1.1: Hierarchical structure of the Browser Object Model (BOM), illustrating the relationship between the `window` object and its sub-objects.

The information can be retrieved from a browser fingerprint includes, screen resolution and color depth, installed fonts, supported media formats, hardware specifications (CPU, GPU, memory), software versions, timezone settings, language preferences, and browser extensions [30]. Research has documented that modern browsers expose hundreds of distinct attributes and behaviors via their JavaScript APIs. Collectively, studies demonstrate that these fingerprinting vectors, encompassing aspects from hardware configuration to graphical rendering, constitute an attack surface enabling the creation of a highly discriminating digital signature for user tracking [35–37].

The appeal of browser fingerprinting lies in its persistence and difficulty to circumvent:

- **Stateless Operation**: Unlike cookies, fingerprinting requires no stored data on the user's device, making it difficult to detect and block [38, 39].
- **Cross-Domain Capability**: Fingerprints can be collected and compared across different websites and domains [23, 30].

[6] https://developer.mozilla.org/en-US/docs/Web/API

- **Resistance to Clearing**: Standard privacy measures like clearing cookies or browsing data do not affect most fingerprinting attributes [40, 41].
- **High Uniqueness**: The combination of numerous attributes often creates highly unique identifiers that can distinguish individual users [29].

However, browser fingerprinting also presents significant challenges and concerns:

- **Privacy Implications**: Users often remain unaware that they are being fingerprinted, as the process operates transparently without requiring user interaction or consent [25, 34]. Studies show that user awareness of fingerprinting techniques remains significantly lower than awareness of cookie-based tracking [42].
- **Variability and Instability**: Browser configurations change over time due to software updates, hardware modifications, new extension installations, or user preference adjustments, creating temporal instability in fingerprints. Research demonstrates that fingerprint stability varies significantly, with some attributes remaining stable for months while others change within days or weeks [43, 44]
- **Computational Complexity**: Modern browsers can expose hundreds of distinct attributes through various APIs, making comprehensive fingerprinting analysis computationally demanding for real-time tracking systems, particularly when processing large-scale datasets [45].
- **Configuration-Fingerprint Relationship**: The relationship between browser configuration parameters and resulting fingerprints remains poorly understood. A large number of elements are not documented, making it difficult to predict fingerprint behavior across different browser configurations [37].

Despite significant research on browser fingerprinting techniques and countermeasures over the past decade several critical gaps remain in our understanding of this privacy challenge [35, 46]. Most notably, systematic exploration of how specific browser configuration parameters affect fingerprint characteristics remains limited in scope. While studies have examined individual fingerprinting vectors [30, 47], comprehensive analysis of parameter-fingerprint relationships across different browser configurations is sparse [43]. This knowledge gap prevents developers from making fully informed decisions about feature implementation and limits users' ability to make evidence-based privacy-preserving configuration choices. Furthermore, existing approaches to analyzing browser fingerprints often rely on methods that may not fully capture the complex interdependencies between configuration parameters and resulting attributes [37, 41]. While some systematic frameworks have been proposed, this limitation can hinder the development of optimal privacy-enhancing technologies and comprehensive fingerprinting countermeasures [33, 44].

1.2 Motivations

The evolution of web browsers, their functionalities, and their customization possibilities, coupled with the increasing automation of tracking, and more specifically

browser fingerprinting, raises questions about the impact of these new possibilities on privacy. Each new personalization and each new option can impact the attributes of the fingerprint, necessitating a deep understanding of the interactions between configuration parameters and the resulting attributes, as well as methods to detect and eliminate these vulnerabilities. The research presented in this thesis is motivated by the need to understand the links between these configuration parameters and the resulting attributes.

1.2.1 Impact of Configurations on Browser Fingerprinting: Fingerprint Variability Across Configurations

A fundamental gap exists in our understanding of how browser configuration parameters impact the browser fingerprint. While extensive research has cataloged various fingerprinting techniques and attributes, we don't have an exhaustive understanding of how specific configuration choices affect the browser fingerprint.

This knowledge gap has several important implications:

- **Developer Blindness**: The developer of browser extension, or a developer of browser didn't have a view of the impact of their new features to the browser fingerprint.
- **User Guidance**: Privacy-conscious users cannot make informed decisions about browser configurations without understanding their fingerprinting implications.
- **Security Assessment**: Organizations cannot properly assess the privacy risks associated with different browser deployment configurations.
- **Countermeasure Development**: Effective anti-fingerprinting tools require detailed knowledge of which attributes are most discriminating and how they relate to underlying configurations.

By systematically exploring the impact of configuration parameters on browser fingerprints, we can:

- Identify which configuration options most significantly affect fingerprint uniqueness
- Provide quantitative measures of privacy risk for different configuration choices
- Enable the development of privacy-preserving default configurations and get a better understanding of the fingerprinting process
- Support the creation of more effective fingerprinting countermeasures with a limitation of side effects

1.2.2 Representation of Browser Fingerprints: Modeling and Representation of Fingerprints

There is no standard method for representing browser fingerprints. These are often saved as JSON files containing key-value pairs. While simple to implement, these approaches suffer from several limitations that hinder large-scale analysis and understanding:

- **Storage Inefficiency**: Redundant storage of identical attributes across multiple fingerprints leads to excessive storage requirements and processing overhead.
- **Lack of Structure**: Flat attribute lists do not capture the hierarchical relationships between browser APIs and their exposed properties.
- **Limited Analysis Capabilities**: Without structured representations for these fingerprints, it is more difficult to perform sophisticated queries, comparisons, and pattern recognition across large fingerprint datasets.
- **Poor Scalability**: Ad-hoc approaches do not scale well to the analysis of thousands or millions of fingerprints across multiple browser versions and configurations.
- **Missing Relationships**: Traditional representations fail to capture the complex dependencies between different fingerprint attributes and their underlying causes.

1.3 Contributions

This work makes several contributions to the understanding and analysis of browser fingerprinting, particularly focusing on the relationship between browser configurations and resulting fingerprint characteristics, as summarized in Figure 1.2.

1.3.1 FP-Rainbow: Analysis of Configuration Impact on Browser Fingerprints

The first major contribution is FP-Rainbow, a systematic approach for exploring and identifying browser configuration parameters that affect the Browser Object Model (BOM) and resulting fingerprints. This contribution addresses the fundamental research question of understanding how individual browser configuration parameters impact fingerprint characteristics. This work has been published in the *Web Conference 2025* (WWW'25) [1].

Key aspects of this contribution include:

- **Systematic Configuration Exploration**: Development of an automated pipeline for systematically testing $1,748$ configuration parameters across 18 versions of Chromium browser

- **Large-scale Dataset Generation**: Creation of a comprehensive dataset containing 61,559 browser fingerprints, representing a systematic exploration of browser configuration
- **Configuration Identification Methodology**: Development of algorithms for reverse-engineering browser configuration parameters from unknown fingerprints, achieving an average identification rate of 78.15%
- **Cross-platform Validation**: Demonstration that datasets generated on single devices can effectively identify configurations across multiple hardware platforms

The practical implications of this contribution include enabling developers to understand the privacy implications of configuration choices, supporting the development of privacy-preserving browser defaults, and providing tools for identifying potentially vulnerable or unusual browser configurations.

1.3.2 Taming the Variability of Browser Fingerprints: Formal Approach to Represent Browser Fingerprint Variability

The second major contribution introduces a novel approach for representing browser fingerprint variability using formal modeling techniques from Software Product Line Engineering (SPLE). This work addresses the need for structured, scalable representations of browser fingerprint data. It was published in the *Systems and Software Product Line Conference* (SPLC'24) [2].

Key innovations include:

- **Feature Model Synthesis**: Development of algorithms for automatically constructing feature models from browser fingerprint datasets, transforming flat attribute-value pairs into hierarchical tree structures
- **Variability Capture**: Formal representation of relationships and constraints between different fingerprint attributes, enabling systematic analysis of fingerprint variability
- **Scalable Storage**: Achievement of 59,000 times reduction in storage requirements compared to traditional JSON-based approaches while preserving complete information
- **Constraint Identification**: Automated identification of mandatory, optional, XOR, and OR relationships between fingerprint features based on their co-occurrence patterns
- **Evolutionary Analysis**: Framework for tracking how browser fingerprints evolve over time as browsers are updated and new features are introduced

This contribution provides a foundation for large-scale fingerprint analysis, enables efficient storage and querying of fingerprint datasets, and facilitates the development of sophisticated fingerprint analysis tools.

1.3.3 BrowserFM: Feature Modeling for Browser Fingerprinting and Configuration Space Exploration

The third contribution, BrowserFM, integrates the systematic configuration exploration of FP-Rainbow with the formal modeling approach used in the previous work, creating a comprehensive framework for browser fingerprint analysis. This work demonstrates the practical applications of feature model-based fingerprint representation. It was also presented at the *Workshop on Measurements, Attacks, and Defenses for the Web* (MADWEB'25) [3].

Key components include:

- **Integrated Analysis Pipeline**: Development of tools that combine configuration exploration with feature model synthesis, enabling comprehensive analysis of configuration-fingerprint relationships
- **Configuration Space Modeling**: Formal representation of the browser configuration space using feature models, capturing the relationships between different configuration parameters
- **Privacy Analysis Tools**: Development of methods for identifying minimal sets of discriminating attributes that can be used to identify configurations parameters
- **Unified Feature Model Representation**: Utilizing feature models to represent browser fingerprints, configuration parameters, and environmental attributes (e.g., hardware, OS), thus creating a unified representation where all contributing factors are modeled as features.

This contribution provides practical tools for researchers and developers, enables privacy-focused browser configuration recommendations, and supports the development of more effective anti-fingerprinting technologies.

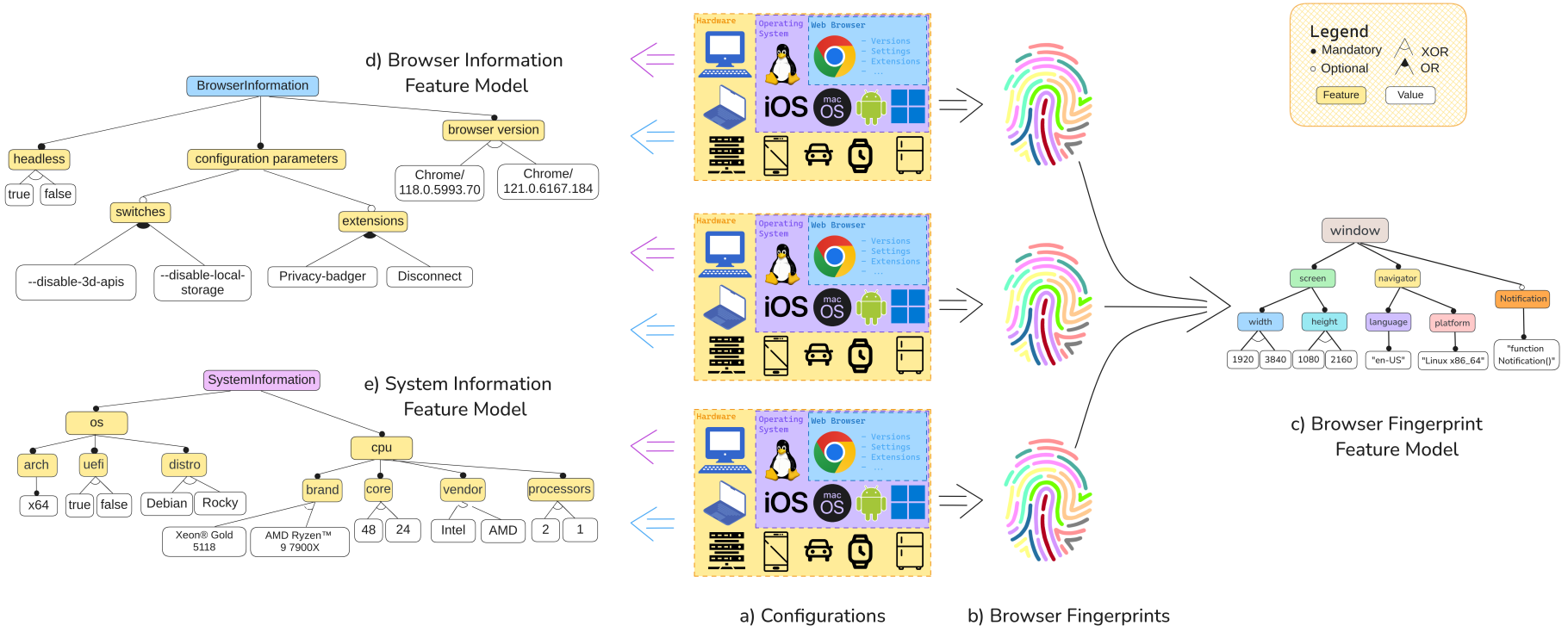


Figure 1.2: **Overview of Contributions**: a) represents the configurations of browsers and their environments, b) is the browser fingerprint generated from their respective configurations, c) is a unified feature model representation of these browser fingerprints, d) is representation of configuration parameters in feature models, and e) the representation of system information in feature models.a) and b) is from FP-Rainbow [1] contribution. c) is from Taming the variation of browser fingerprints [2]. d) and e) are from the BrowserFM [3].

1.4 Manuscript Organization

This thesis is organized into six chapters, each building upon the previous work to develop a comprehensive understanding of browser fingerprinting and configuration analysis.

Chapter 1: Introduction provides the context, motivation, and problem statement for the research. It defines key concepts, outlines the research questions, and summarizes the main contributions of the thesis.

Chapter 2: Background presents a comprehensive review of the state of the art in browser fingerprinting, web tracking, and software product line engineering. This chapter establishes the theoretical foundations and identifies gaps in existing research that motivate the proposed approaches.

Chapter 3: Large-Scale Analysis of Configuration Impact on Browser Fingerprints introduces FP-Rainbow, the systematic approach for exploring browser configuration parameters and their impact on fingerprints. This chapter presents the methodology, experimental setup, and results from analyzing $1,748$ configuration parameters across multiple browser versions.

Chapter 4: Formal Representation of Browser Fingerprint Variability presents the feature modeling approach for representing browser fingerprints. This chapter details the algorithms for building and merging feature trees, the synthesis of feature models from fingerprint datasets, and the practical applications of this formal representation.

Chapter 5: Tools, Datasets, and Platforms describes the experimental infrastructure, datasets, and tools developed as part of this research. This chapter provides detailed information about reproducibility, data availability, and the technical platforms used for large-scale fingerprint collection and analysis.

Chapter 6: Conclusions and Perspectives summarizes the main findings, discusses the implications of the research, and outlines future research directions. This chapter also reflects on the broader impact of the work and its potential applications in privacy protection and browser development.

All datasets, source code, and experimental tools described in Chapter 5 are made available as open-source resources to support reproducibility and enable further research in this important area.

Chapter 2

Background

2.1 User Tracking

Prior to the emergence of the web, various forms of tracking and tracing individuals already existed throughout history [48, 49]. Historically, the monitoring of political figures, as well as ordinary citizens, was common practice, particularly through espionage and state-organized surveillance [50]. In Venice, the Council of Ten implemented a system of anonymous denunciation and surveillance of diplomats and influential citizens [51]. During World War II, governments utilized registries and identity papers to monitor population movements [52]. In authoritarian regimes, filing systems were established to track political opponents or specific communities, as exemplified by the Stasi in East Germany [53] and Vichy France's surveillance apparatus [54]. During the Cold War era, intelligence services gathered information on citizens through telephone tapping or mail surveillance [55]. Even in the commercial domain, merchants maintained records of their customers to better target their offerings [56]. These examples illustrate that individual tracking is not a new phenomenon, but that the techniques have evolved with the advent of the web [57, 58].

2.1.1 Web Tracking history

The evolution of web tracking technologies since the 1990s represents a dynamic arms race between data collection practices and privacy protection measures, with periods of rapid innovation alternating with regulatory responses [21, 23, 59]. From basic server-side IP address logging in the early 1990s [60, 61] to the complex ecosystem of third-party cookies [62], fingerprinting [29, 30], and cross-site tracking technologies [23, 63], these developments have significantly expanded the scope and precision of user behavior monitoring across the internet [40, 64]. Understanding this technological and regulatory progression is essential to comprehend the current state of online privacy [65, 66] and the ongoing challenges in balancing personalized web experiences with user data protection [67, 68].

IP Address

With the arrival of the web, user tracking techniques evolved, notably enabling automatic identification via IP addresses [64, 69, 70]. In a security context, the IP address is used to verify the identity of users accessing sensitive services [69]. Web server logs traditionally use IP addresses as the primary method for identifying and counting unique visitors, despite known limitations in accuracy due to shared connections, proxy servers, and dynamic IP allocation [71, 72]. This technique has been exploited to identify users accessing illegal content, particularly through IP address analysis on peer-to-peer networks, and even through the TOR network [73]. In France, the Hadopi law was established in 2009 to combat illegal downloading by identifying the IP addresses of internet users sharing or accessing files protected by copyright, before receiving a notification letter at their home [74–76] To circumvent this type of surveillance, many users resort to VPNs or proxies in order to mask their true IP address [77].

Cookies first party

Cookies were first introduced in 1994 by Lou Montulli, an engineer at Netscape Communications, as part of the development of the Netscape Navigator browser [78–81]. A cookie is a small text file stored by the web browser on the user's computer at the request of a website. Technically, when a user visits a site, the server may send an HTTP `Set-Cookie` instruction in the response header. The browser then records this cookie locally and will automatically return it to the server during subsequent requests to that same site, via the HTTP `Cookie` header. This innovation was designed to enable websites to store client-side information in order to manage user sessions and customize the browsing experience. The first documented use of cookies occurred on the MCI e-commerce site, enabling users shopping cart contents to be stored directly in the browser rather than on the site's server [82]. As early as 1997, the cookie specification was formalized in RFC 2109 [20]. It was subsequently extended in RFC 2965 in 2000 [62], then in RFC 6265 in 2011 [22], with a new version currently under development [83]. The rapid adoption of cookies by major web browsers and websites marked the beginning of their widespread use for tracking and personalization on the Internet. When creating cookies, Lou Montulli and the Netscape team had rejected a solution based on assigning a unique and permanent identifier to each browser in order to protect user privacy. However, cookies have been repurposed to track users even when they change IP addresses.

Cookies third party

The use of third-party cookies has rapidly become widespread for tracking users across different websites. A third-party cookie is a cookie present on a website but belonging to a different domain than the one visited by the user. In 2016, their number was twice that of first-party cookies [84]. These cookies allow the integration of external functionalities on a website. When a site integrates a Facebook "Like" button or a Disqus comment module, a third-party cookie is set by these platforms during page loading, even without user interaction [21, 64]. This practice allows

these services to collect information about users' browsing habits across different websites. Advertising networks extensively exploit this technique by placing third-party cookies on numerous partner sites, enabling them to track user navigation and build detailed profiles of their interests [23]. A large-scale study conducted by Englehardt and Narayanan on one million websites revealed the extent of this cross-site tracking practice for behavioral advertising and audience analysis [23]. This cross-site tracking capability raises important privacy concerns, as it allows companies to create detailed user profiles without their knowledge [65]. Faced with these challenges, many regulations, such as GDPR in Europe, now require obtaining user consent before setting third-party cookies.

Legal aspects

In 2018, Europe implemented the General Data Protection Regulation (GDPR), which mandates enhanced protection of personal data and user rights online [85]. Cookie consent requirements originated from the ePrivacy Directive (2002) and were strengthened by the 2009 amendments [86, 87], though the specific implementation through consent banners was developed by the industry to comply with these legal obligations [27]. These measures have enabled users to become more aware of the impact of cookies on their privacy [27, 28, 88]. The GDPR has introduced general measures for data protection, affecting how cookies containing personal data are processed [89, 90]. However, studies have shown that many websites employ dark patterns in their consent dialogs, making it difficult for users to exercise genuine control over their data [91]. Despite the regulatory framework, the effectiveness of cookie consent mechanisms remains questionable, with many implementations failing to provide meaningful user choice or have problems with implementation [89, 92, 93].

The GDPR framework permits fines up to €20 million or 4% of global annual turnover for serious violations [85, 94]. While GDPR and similar privacy regulations provide legal frameworks for user protection, empirical studies have identified significant gaps in practical enforcement and user protection [88, 91]. These limitations have prompted the development of technical countermeasures, including browser-based privacy tools and industry initiatives.

Defenses mechanisms

Major browsers and platforms have been adapting to this changing landscape. Google initially announced plans in 2020 to phase out third-party cookies in Chrome [95], forcing advertisers and tracking companies to develop new tracking methods through the Privacy Sandbox initiative [96]. However, in July 2024, Google reversed this decision and announced it would not be phasing out third-party cookies in Chrome. Instead, Google decided to introduce a new user choice experience that would allow Chrome users to make informed decisions about third-party cookie usage across their browsing experience, while continuing to develop Privacy Sandbox technologies as alternative solutions for advertisers and publishers [96].

Today, many web browsers like Brave, Firefox, and Tor have implemented various defense mechanisms to protect user privacy and prevent tracking [97]. Pan *et al.* [98] have shown that Chrome's anti-tracking capabilities have increased between versions

83 and 90. A number of web browsers have adopted private browsing modes [99], and some of them have also adopted features like cookie isolation at different levels. Brave has implemented cookie isolation per domain, while Firefox 86 has implemented cookie isolation per website [100]. Other web browsers like Tor provide anonymity and privacy protection by at network level, utilizing onion routing technology [101, 102]. Additionally, new privacy-focused browsers such as LibreWolf,[1] Waterfox,[2] Floorp,[3] Trivalent,[4] IridiumBrowser,[5] and Brave[6] have emerged, offering users more control over their online experience.

The community has also developed browser extensions and plugins to enhance privacy and security, such as uBlock Origin,[7] Ghostery,[8] NoScript,[9] Privacy Badger,[10] and Disconnect[11] [38, 103, 104]. These tools are based on different techniques, such as blacklisting known tracking domains, using heuristic algorithms, or blocking JavaScript execution entirely [38, 103].

The development of these privacy protection mechanisms has been driven by increasing concerns about web tracking technologies [105]. Understanding the effectiveness and implementation of these various approaches is crucial for maintaining user privacy in the modern web environment.

2.2 Browser Fingerprint

As traditional tracking mechanisms like cookies face increasing restrictions from browsers and privacy regulations, browser fingerprinting has emerged as a sophisticated alternative for user identification and tracking. This technique exploits the inherent diversity in users' software and hardware configurations to create unique digital signatures that persist across browsing sessions without requiring stored identifiers. By leveraging the rich information exposed through web APIs and JavaScript execution environments, fingerprinting can achieve remarkably high levels of user uniqueness.

A browser fingerprint is a unique set of attributes (keys) and their corresponding values (see Table 2.1), generated locally within the user's web browser and transmitted to the server, where it is stored and remains outside of the user's control. Browser fingerprints generally consist of structured attribute-value pairs that characterize browser properties or, quite similarly, of function calls and their return values. Browser fingerprints originate from the Browser Object Model (BOM).[12] Not to be confused with the Document Object Model (DOM), which focuses on the

[1] https://librewolf.net/
[2] https://www.waterfox.net/
[3] https://floorp.app/
[4] https://github.com/secureblue/Trivalent
[5] https://iridiumbrowser.de/
[6] https://brave.com/
[7] https://ublockorigin.com/
[8] https://www.ghostery.com/
[9] https://noscript.net/
[10] https://privacybadger.org/
[11] https://disconnect.me/
[12] https://en.wikipedia.org/wiki/Browser_Object_Model

content of a web page, the BOM is a programming interface to interact with the browser and provides access to objects, attributes, properties and methods related to the browser's window. BOM enumeration systematically explores the browser's public APIs to obtain all exposed attributes, values and methods from the browser, providing a comprehensive snapshot of the browser and its configuration.

This section examines the technical foundations of browser fingerprinting, from its origins in early JavaScript implementations to modern advanced techniques that exploit hardware characteristics, explores the various software and hardware attributes that contribute to fingerprint uniqueness, and analyzes the ongoing arms race between fingerprinting methods and privacy protection measures.

Table 2.1: Example of Browser Fingerprint Attributes and Values

Attribute Name	**Value**
`navigator.userAgent`	`Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.5938.149 Safari/537.36`
`screen.width`	`1920`
`screen.height`	`1080`
`navigator.language`	`en-US`
`navigator.platform`	`Linux x8_64`
`navigator.cookieEnabled`	`true`
`Notification`	`function Notification()  [native code]`

2.2.1 History and evolution

The first implementation of JavaScript was realized in 1995 by Brendan Eich, then engineer at Netscape Communications [106, 107]. This language, initially called Mocha then renamed LiveScript before becoming JavaScript, was integrated for the first time in the Netscape Navigator 2.0 browser. The objective was to enable the execution of client-side scripts in order to make web pages interactive, complementing static HTML. This innovation quickly allowed developers to access and manipulate HTML document elements and browser objects (concepts later formalized as DOM and BOM), dynamically modify page content, and interact with users without requiring a complete page reload. The integration of JavaScript in Netscape marked a turning point in web evolution, paving the way for dynamic web applications and the emergence of new tracking techniques, such as browser fingerprinting, which exploit the capabilities offered by this language to collect information about the user's software and hardware environment.

For instance, screen size is accessible through the `window.screen.width` and `window.screen.height` attributes, and the language through `window.navigator.language`, which provide values such as `1920`, `1080`, and `en_US`, respectively. These values allow developers to adapt their applications to the device, but can also be used to identify the browser, or more specifically, the device. The risk increases as more attributes are considered.

Browser fingerprinting encompasses a set of techniques allowing the collection of information exposed by the user's browser and operating system. This information includes browser version, language, screen resolution, installed fonts, plugins, etc. [35,108] From this information, it is possible to generate a unique or quasi-unique browser fingerprint [29]. This approach exploits the fact that the combination of these attributes, while each taken individually is poorly discriminating, allows obtaining a very specific signature for each user [34, 109]. A detailed classification of different browser fingerprinting techniques has been proposed by Upadhyaya *et al.* [110], who notably distinguish methods based on software and hardware attributes.

One of the first major academic studies on the subject is that of Peter Eckersley, published in 2010 under the title *How Unique Is Your Web Browser?* [29]. In this study, the author shows that the majority of browsers can be uniquely identified from their configuration. He uses JavaScript scripts to collect information about the browser environment. The Panopticlick project, launched by the Electronic Frontier Foundation (EFF), demonstrated the power of fingerprinting on a large scale.[13] This project collected fingerprints from hundreds of thousands of users [29]. Subsequent work, such as that of Acar *et al.* [111], identified and mapped the use of fingerprinting on the web at large scale, revealing that many third-party sites exploit these techniques for advertising tracking or fraud prevention purposes. Other work by the same author [30], also highlighted the diversity and persistence of tracking mechanisms, including fingerprinting, used to circumvent restrictions imposed on traditional cookies.

The arrival of technologies like Flash or Java also facilitated the collection of information about the user's system [23]. This opened the way to more sophisticated fingerprinting methods [112, 113]. Research by Acar *et al.* [30] and Englehardt *et al.* [23] highlighted the sophistication of browser fingerprinting techniques employed by online entities. These techniques combine attributes to create a unique fingerprint, enabling user tracking across different sites, even when attempting anonymity through tools like VPNs or incognito mode.

With the evolution of web standards (HTML5, WebGL, Canvas, AudioContext, etc.), the attack surface for fingerprinting has considerably expanded [33, 37, 114]. State-of-the-art research shows that modern browser fingerprints can include over 13,000 distinct attributes [37], with some findings indicating upwards of 16,000 attributes, highlighting the growing complexity of fingerprinting. A significant privacy concern stems from the accessibility of these fingerprinting attributes through standard browser APIs, which do not require explicit user consent, enabling the stealthy collection of fingerprinting data.

2.2.2 Software attributes

Modern web browsers expose numerous software-based attributes that can be collected for fingerprinting purposes. JavaScript provides access to system information including browser version, user agent strings, supported MIME types, and available APIs [34, 113].

[13] https://panopticlick.eff.org/

Historically, browser plugins such as Flash and Java enabled comprehensive font enumeration and system probing capabilities. Eckersley *et al.* [29] demonstrated that among browsers supporting Flash or Java, each browser presented at least 18.8 bits of identifying information, with 94.2% of browsers being unique in their sample. However, the deprecation of Flash (end-of-life December 2020) and the decline of Java browser plugins have significantly altered the fingerprinting landscape [45].

Browser Configuration Attributes. Modern fingerprinting techniques collect detailed configuration information including user agent strings, accepted languages, timezone settings, screen resolution, color depth, and enabled JavaScript features. These attributes provide insights into user preferences, system locale settings, and hardware capabilities [35, 45].

Extension and Plugin Fingerprinting. Browser extensions constitute a significant fingerprinting vector despite privacy countermeasures. Starov *et al.* [115] demonstrated that extensions can be enumerated through resource loading patterns and API behavior analysis. Sjösten *et al.* [116] used a different approach by detecting extensions through their web accessible resources. All of these techniques create unique signatures based on the user's installed extensions.

Font-based Fingerprinting. Font enumeration represents a persistent fingerprinting vector that has evolved with browser security measures. Fifield *et al.* [112] demonstrated font-based identification through rendering metrics and glyph measurements. Despite browser restrictions on direct font enumeration, modern techniques exploit Canvas API and CSS font-matching behaviors [33, 113]. Acar *et al.* [30] showed that font fingerprinting can achieve high entropy values. Entropy quantifies the uniqueness of an attribute in browser fingerprint, where higher entropy of an attribute indicates a greater likelihood that a specific browser configuration is unique among a population of users.

JavaScript Engine Characteristics. JavaScript execution environments provide multiple fingerprinting opportunities through engine-specific behaviors, performance characteristics, and API implementations. Research by Mulazzani *et al.* [117] demonstrated that different JavaScript engine implementations, with their distinct optimization strategies and behavioral patterns, create detectable signatures that enable reliable browser identification [37]. Additionally, WebGL and Canvas API implementations exhibit engine-specific behaviors that can be exploited for identification [47, 114].

2.2.3 Hardware attributes

Browser fingerprinting is not only impacted by software, it can also be directly influenced by hardware. Modern hardware has become very complex, and the work of [118] has shown that even with the same GPU, the browser fingerprint can differ, and the GPU can be uniquely identified.

Other works, such as [43, 114], show that the WebGL API and Canvas used to render images can be used to uniquely identify hardware. Not only the GPU can be uniquely identified, but also the CPU. Saito *et al.* [119] proposed a method to estimate CPU features using browser fingerprinting. Their approach involves running carefully crafted JavaScript code in the browser to measure the execution time

of specific operations, such as arithmetic and floating-point calculations, memory access, and other low-level tasks [120]. By analyzing the timing results, they are able to infer detailed characteristics of the user's CPU, including its architecture, clock speed, and the presence of certain instruction sets. This technique allows for the remote identification of CPU features without requiring any explicit user permission or access to native code, thereby increasing the uniqueness and stability of the browser fingerprint. Trampert *et al.* [121] presented a browser-based CPU fingerprinting technique.

More suprisingly Chalise *et al.* [122] demonstrated than the Speaker can impact the browser fingerprinting through the Web Audio API. They found that, unlike other fingerprinting vectors, audio fingerprints can be unstable, with some browsers producing different fingerprints across repeated attempts. However, by applying a graph-based analysis that aggregates all fingerprints produced by a browser, they were able to create a highly stable fingerprinting mechanism. Their results also show that audio fingerprints are less diverse than other vectors, with only 95 distinct fingerprints among 2093 users. Nevertheless, combining web audio fingerprinting with other techniques can significantly improve identification: for example, adding audio fingerprinting to Canvas fingerprinting increases entropy by 9.6%. Their findings also challenge current W3C security and privacy recommendations regarding audio fingerprinting. Englehardt *et al.* [23] demonstrated that audio fingerprinting is already being used to track users with the *audiocontext*.

2.2.4 Advanced techniques

Gómez-Boix *et al.* [45] analyzed the effectiveness of large-scale fingerprinting and confirmed that the combination of multiple attributes allows for high uniqueness, making the technique particularly effective for user tracking.

Multi-attribute Fusion and Correlation. Advanced fingerprinting techniques leverage sophisticated algorithms to combine multiple attributes in ways that maximize uniqueness while maintaining stability. Vastel *et al.* [123] demonstrated that effective fingerprinting requires careful attribute selection and weighting to create robust identification mechanisms that resist common evasion techniques.

Cross-Device Fingerprinting. Advanced techniques extend beyond single-device identification to track users across multiple devices. Zimmeck *et al.* [124] analyzed cross-device tracking methods that combine browser fingerprints with other identification vectors to create persistent user profiles across different browsers and devices.

Service Worker Exploitation. Although not strictly related to browser fingerprinting, the work of Karami *et al.* [125] on the exploitation of Service Workers represents an adjacent technique for inferring users sensitive information such as web history and behavioral patterns. Service Workers provide persistent execution contexts that can be exploited for long-term tracking and data collection. w
WebSocket and Network Probing. Advanced fingerprinting techniques leverage network-level probing to gather additional identifying information. Erkkilä *et al.* [126] explored the potential of WebSocket port scanning for gathering system information, while Nibert *et al.* [127] investigated WebRTC protocol exploitation

for IP address leakage and network topology inference.

2.2.5 Defense measures

The proliferation of browser fingerprinting techniques has prompted the development of various countermeasures aimed at protecting user privacy. These defense mechanisms can be broadly categorized into several approaches, each with distinct advantages and limitations.

Efforts to counter browser fingerprinting have resulted in the development of diverse anti-fingerprinting methods and techniques [128]. These anti-fingerprinting tools are primarily focused on manipulating the attributes utilized in fingerprinting to reduce the uniqueness of user fingerprints and bolster privacy. By altering and obfuscating specific attributes, these tools seek to make it more challenging for online entities to track and identify users based on their browser fingerprints. However, these tools can impact the user experience and break some websites. They can also be recognizable and are difficult to maintain over time [113].

Browser Extensions and Add-ons. Torres *et al.* [41] proposed FP-Block, an approach that distinguishes regular (intra-domain) tracking from cross-domain tracking, generating different web identities for each site where a third party is embedded, thus preventing tracking across the web while maintaining legitimate functionality. Several browser extensions have been developed like mentioned in section 2.1.1 to counter fingerprinting attacks. However, different studies have demonstrated that many countermeasure extensions can introduce subtle side-effects that make browsers more identifiable, rendering the extensions counterproductive [115]. The work of Vastel *et al.* [123] with the FP-TESTER framework provides automated testing capabilities to evaluate the effectiveness of browser fingerprinting countermeasure extensions, helping developers identify and mitigate unintended consequences.

Attribute Spoofing and Randomization. One common approach involves spoofing or randomizing fingerprinting attributes to reduce user uniqueness. Baumann *et al.* [129] developed a disguised Chromium browser that provides robust protection against browser, Flash, and Canvas fingerprinting without deactivating browser features. Their approach uses a large dataset of real-world configurations to mask actual system properties while maintaining usability, achieving successful fingerprinting protection. Obidat [130] presented Canvas Deceiver, a novel defense mechanism specifically targeting Canvas fingerprinting. Unlike existing approaches that rely on randomization, Canvas Deceiver employs a deterministic algorithm that avoids detection while providing effective anonymity. Testing showed that it could increase a user's anonymity set from 634 to 7,847 people in the Browserleaks dataset.

Moving Target Defense Strategies. Laperdrix *et al.* [131] proposed a multi-level reconfiguration and diversification approach that leverages software diversity for privacy protection. Their strategy treats constant change as a moving target defense against fingerprint tracking, breaking the essential property of stability over time that fingerprinting relies upon. This approach demonstrates how the same diversity that enables fingerprinting can be harnessed to defeat it through automatic reconfiguration.

Client-side Diversification. Trickel *et al.* [132] developed client-side diversifi-

cation techniques specifically for defending against extension fingerprinting. When a browser extension modifies the DOM, this extension can be clearly identified [115]. They proposed a tool that randomizes the id modified by the browser extension in the DOM elements to make it harder for fingerprinters to identify specific extensions.

Detection and Response Systems. Advanced defense strategies focus on detecting fingerprinting attempts rather than preventing them entirely. Iqbal *et al.* [46] developed machine learning-based approaches to automatically detect browser fingerprinting behaviors.

Deception-based Approaches. Nikiforakis *et al.* [133] introduced PriVaricator, a system that deceives fingerprinters by providing deliberately false information. This approach recognizes that completely blocking fingerprinting attempts may be detectable, while providing plausible but incorrect information can confuse tracking systems without triggering defensive responses.

Evaluation and Testing Frameworks. Salomatin *et al.* [128] conducted comparative studies of various countermeasures against browser fingerprinting, examining their impact on reducing user uniqueness. Their research analyzes different defensive approaches including browser changes and specialized plugins that can block access to browser attributes or replace their values with randomized ones.

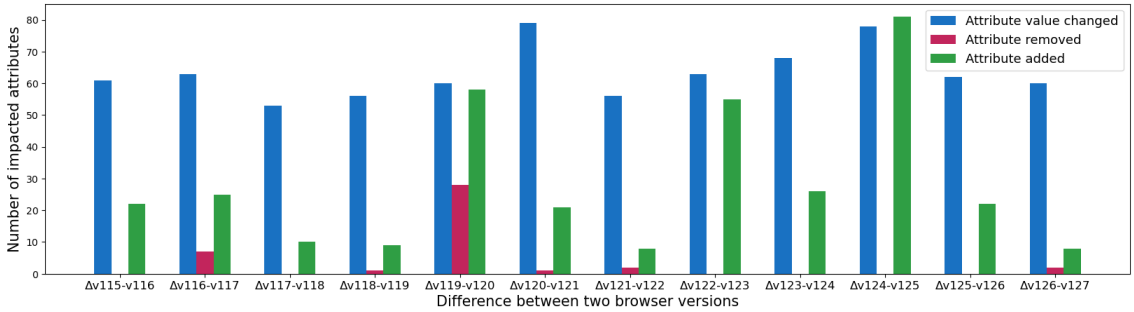
2.2.6 Limitations and Challenges

Despite various approaches, effective defense against browser fingerprinting remains a significant challenge [35, 134]. Many countermeasures suffer from detectability issues, where the presence of the defense mechanism itself becomes a fingerprinting attribute [113]. Indeed, any attempt to spoof or block information can create an anomaly that makes the browser even more unique.

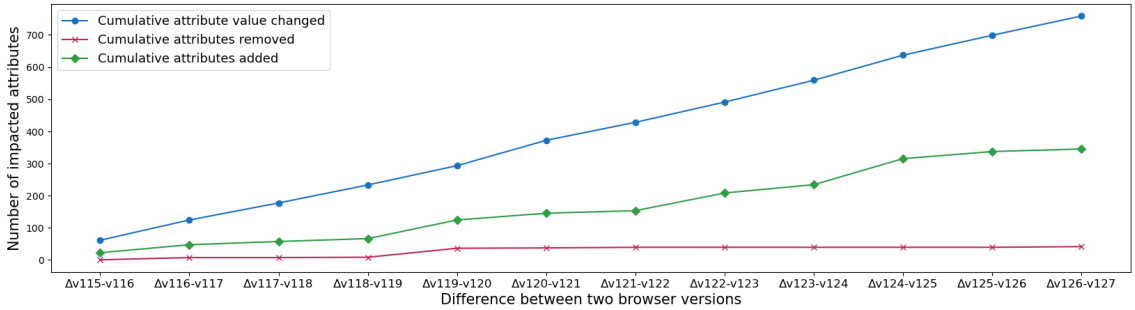
Furthermore, the arms race between fingerprinting techniques and defenses means that static solutions quickly become obsolete as attackers adapt their methods [46]. The effectiveness of fingerprinting defenses also depends heavily on adoption rates. Individual solutions may provide limited protection if they create small, easily identifiable user populations, defeating the goal of blending in with the crowd [29].

Despite various defense approaches proposed in the literature [133], browser vendors, academics, and standards bodies continue to struggle with providing meaningful fingerprinting protections [35, 46]. Browser fingerprinting defense mechanisms continue to evolve in response to increasingly sophisticated tracking techniques.

While no single approach provides complete protection, the combination of multiple defensive strategies, coupled with broader adoption of privacy-focused technologies, offers the best current protection against fingerprinting-based tracking. The ongoing development of these techniques reflects the dynamic nature of web privacy, where technical innovations must continuously adapt to emerging threats. These defenses represent a delicate trade-off between privacy and user experience, requiring careful consideration of user needs and technological limitations, especially given the modern web's deep dependency on JavaScript for core functionality [135].



(a) Attribute differences between consecutive browser versions.



(b) Cumulative attribute differences over consecutive browser versions.

Figure 2.1: Attribute evolution of Chromium's default configuration from versions 115 (baseline) to 127. Many attributes are added and values change, while few are removed.

2.2.7 Configuration Impact and Fingerprint Evolution

The variability in browser configurations presents complex challenges in terms of user privacy and security. Browser configurations (from *switches*,[14] thousands of configuration parameters are available[15]) significantly impact browser fingerprints. As presented in the survey proposed by Pereira *et al.* [136], many studies focus on exploring configurations, *e.g.*, for performance prediction [137], optimization [138] or dynamic reconfiguration [139].

Fingerprints change as the browser evolves [113]. As depicted in Figure 2.1a, even with the same configuration, BOM attributes are added, removed, or their values changed between browser versions. Over 13 versions, as shown in Figure 2.1b, we observed many new attributes being added, few being removed. Various studies have examined the potential of browser configurations, their code [140], and standard updates to reduce the effectiveness of browser fingerprinting techniques. Researchers have explored the impact of changes in browsers and Web standards on fingerprinting resistance [129].

Despite progress in understanding and addressing browser fingerprinting, several limitations persist in existing approaches like the browser's complexity, its wide

[14] https://peter.sh/feed/chromium-command-line-switches/
[15] https://chromium.googlesource.com/chromium/src/+/main/docs/configuration.md

range of uses and its tendency to evolve rapidly. There is thus a critical need to thoroughly explore browser configurations to pinpoint configurations and settings that may pose privacy or security risks, as well as provide tools to assist browser developers in minimizing the side-effects of their code on the Browser Object Model (BOM).

2.2.8 Applications and Use Cases

Various use cases can leverage browser fingerprints beyond tracking. The usage of browser fingerprinting has been increasing over time due to various factors, including for security, advertising, attacking, as well as for substituting or complementing the use of cookies due to increased user awareness about them and third-party cookie deprecation. Browser fingerprinting is increasingly adopted by websites for tracking and user identification [23], and interestingly, for bot detection [141, 142].

For testing and debugging purposes, developers could randomly select and execute browser configurations that are representative of their user bases, or even attempt to approximate configurations that exhibit bugs. Anomalies and inconsistencies in fingerprints [113] may indicate potential security threats, *e.g.*, attributes that indicate bots or crawlers. Moreover, by analyzing the characteristics of browser fingerprints, it is possible to identify the most discriminating attributes, *i.e.*, rare attributes, attributes specific to some configurations, or attributes with high entropy. Identifying these would help developers reduce privacy risks. Finally, to manage the complexity of the BOM, the thousands of sources of variability, and the speed at which browser's evolve, developers require new approaches and automated support to analyze browser fingerprints.

Formal Representation and Analysis Approaches. Recent research has explored formal methods for representing and analyzing browser fingerprints. Schwarz *et al.* [37] propose a framework for BOM enumeration, the creation of templates for browser fingerprints and their comparison. However, their framework's cleaning phase is based on the difference after a refresh of the web page, and doesn't include the other dynamic attributes in longer terms. During the analysis phase, they compare browser fingerprints one by one, which is not scalable at large scale and limits the usage. Andriamilanto *et al.* [143] propose a tool to select the best attributes to identify users. However, the experimentation is limited to a handful of attributes, and doesn't take into consideration all other existing attributes with potentially higher entropy. These studies highlight both the versatility of feature models across different domains and their potential applications to browser fingerprinting scenarios.

2.3 Software Product Line Engineering (SPLE)

Software Product Line Engineering (SPLE) is a software engineering paradigm that focuses on developing families of related software systems, rather than individual systems in isolation [144]. This approach leverages commonalities among system variants while managing their differences through systematic variability management [145, 146]. The fundamental principle underlying SPLE is the recognition that many software systems share common features and functionality, making it economically advantageous to develop them as a coordinated family rather than as independent products [147]. SPLE has gained significant traction in various domains due to its potential for reducing development costs, improving time-to-market, and enhancing software quality through systematic reuse [148–150]. The approach becomes particularly valuable when organizations need to maintain multiple variants of similar software systems, each tailored to specific market segments, deployment environments, or functional requirements [151, 152].

2.3.1 Fundamental concepts

The foundation of SPLE rests on several key concepts that distinguish it from traditional software development approaches. At its core, SPLE recognizes that software systems within a domain often exhibit both commonalities and variabilities that can be systematically managed [144, 150].

Domain Engineering and Application Engineering. SPLE follows a two-phase development process [144]. Domain engineering focuses on analyzing the problem domain to identify commonalities and variabilities among potential system variants, developing reusable assets that can be shared across the product line. Application engineering then leverages these reusable assets to derive specific product variants according to particular requirements [149]. This separation allows organizations to amortize the cost of developing common functionality across multiple products while maintaining the flexibility to create specialized variants.

Variability Management. Central to SPLE is the systematic management of variability, which involves the ability of a software system to be customized for different contexts and requirements [146]. Variability management involves identifying variation points in the software architecture, documenting the available choices at each point, and providing mechanisms for selecting and implementing specific configurations [150]. This requires careful consideration of dependencies between different variation points and the constraints that govern valid combinations of choices.

Feature Models and Configuration Spaces. Feature models serve as a key notation for documenting and managing variability in software product lines [146]. They provide a hierarchical representation of features and their relationships, enabling systematic reasoning about valid product configurations [153]. Modern software systems present vast configuration spaces that require sophisticated modeling and analysis techniques to ensure valid and optimal configurations.

Reusable Assets and Platform Development. SPLE emphasizes the development of reusable software assets that can be shared across product variants [144, 149]. These assets may include source code components, architectural pat-

terns, test cases, documentation, and other development artifacts. The platform serves as the foundation for product derivation, providing both the common functionality shared by all products and the variation mechanisms needed to create specific variants.

Economic Benefits and Industrial Experience. The economic advantages of SPLE have been documented in industrial practice through multiple case studies. Pohl *et al.* [150] provide foundational analysis of the economic principles underlying software product line engineering. Van der Linden *et al.* [149] present comprehensive industrial experiences and best practices for product line engineering implementation, including economic considerations for return on investment. The economic benefits of SPLE stem from systematic reuse principles, as established by Bosch [151] in his analysis of product-line architectural approaches.

Quality Benefits Through Systematic Reuse. SPLE approaches contribute to improved software quality through systematic reuse and standardized development processes. Clements and Northrop [144] establish that defects found and fixed in core assets benefit all products in the line, supporting higher overall quality compared to independent development approaches. The transition to systematic reuse patterns, as discussed by Krueger [152], enables more effective maintenance strategies across product families.

2.3.2 Feature modeling

Feature modeling represents one of the most successful and widely adopted techniques in SPLE for capturing and managing variability. Feature models provide an intuitive tree-like notation for representing the features of a software system and the relationships between them [146, 154, 155].

Feature Model Structure and Semantics. A feature model organizes features in a hierarchical structure where parent features represent more abstract functionality and child features provide refinements or specific implementations. The model includes various types of relationships (see Figure 2.2): mandatory features that must be included whenever their parent is selected, optional features that may or may not be included, alternative features where exactly one must be chosen from a group, and or-features where one or more may be selected [155]. Cross-tree constraints add additional expressiveness by capturing dependencies and conflicts between features that cannot be represented through the tree structure alone [155].

Feature Model Synthesis and Reverse Engineering. Creating feature models for existing systems presents significant challenges, particularly for legacy systems or large-scale software with complex configuration spaces. Several approaches have been developed to address this problem. She *et al.* [156] proposed efficient algorithms for synthesizing feature models from various sources of variability information. Lopez-Herrejon *et al.* [157] explored the use of evolutionary algorithms for reverse engineering feature models, while Haslinger *et al.* [158] focused on extracting feature models from sets of valid feature combinations.

Feature Model Evolution and Maintenance. Feature models, like other software artifacts, evolve over time as new features are added, existing features are modified, or requirements change. Acher *et al.* [159] developed techniques for

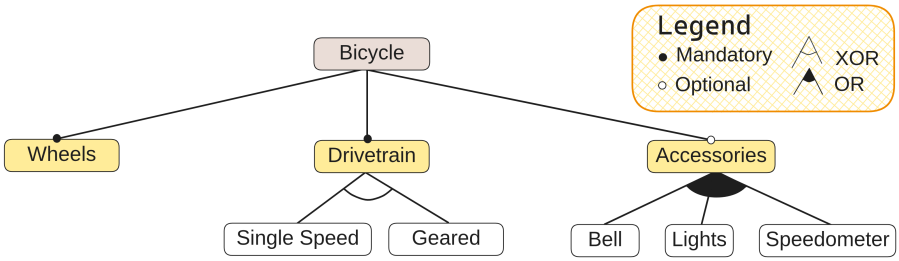


Figure 2.2: **Example of a Feature Model for a Bicycle:** The legend (top right) illustrates the notation for feature relationships: mandatory features (black dot), optional features (white dot), exclusive-or (XOR) groups (curved arc), and inclusive-or (OR) groups (filled arc).

supporting reverse engineering and maintaining feature models throughout their lifecycle. Dintzner *et al.* [160] analyzed feature model changes in large-scale systems like the Linux kernel, demonstrating the practical challenges of managing feature model evolution in real-world scenarios.

Automated Analysis and Reasoning. Feature models enable various forms of automated analysis that support product line development and maintenance. Benavides *et al.* [153] provided a comprehensive survey of automated analysis techniques for feature models, covering satisfiability checking, dead feature detection, core feature identification, and configuration optimization. These techniques rely heavily on constraint satisfaction and Boolean satisfiability solving to reason about large configuration spaces efficiently.

Configuration Sampling and Testing. The exponential nature of configuration spaces makes exhaustive testing impractical for most real-world systems. Kaltenecker *et al.* [161] proposed distance-based sampling techniques for software configuration spaces that aim to select representative configurations for testing while managing the computational cost. Xiang *et al.* [162] developed probability-aware diversification strategies that work with SAT solving to generate diverse configurations efficiently.

Performance Prediction and Optimization. Understanding the performance implications of different feature combinations is crucial for product line engineering. Guo *et al.* [137] developed variability-aware performance prediction approaches using statistical learning techniques. Canales *et al.* [163] demonstrated how feature models can be combined with genetic algorithms to optimize complex configuration spaces, such as Java Virtual Machine flags, showing practical applications of SPLE techniques to system tuning and optimization.

Feature Model Synthesis. The automatic generation of feature models from existing data has emerged as a crucial research area within SPLE. Ryssel *et al.* [164] employed concept analysis to generate feature models that incorporate or-groups and xor-groups, automatically extracting meaningful feature relationships from existing configurations. Similarly, She *et al.* [156] tackled the problem of automatic model synthesis from propositional constraints, proving its NP-hardness and proposing enhanced formulas in both conjunctive normal form (CNF) and disjunctive normal form (DNF). The application of evolutionary algorithms to reverse engineer feature

models has been explored by Lopez-Herrejon *et al.* [157], providing valuable insights into the potential for reconstructing feature models from existing systems. Acher *et al.* [165] proposed a systematic procedure for extracting feature models from product descriptions, enabling a more structured and efficient modeling process. In the domain of requirements engineering, Weston *et al.* [166] introduced a comprehensive framework to support software product line engineers in constructing feature models based on natural language requirements. Additionally, She *et al.* [167] presented a novel procedure that simplifies the reverse engineering process of feature models, introducing a ranking heuristic to identify parent candidates and automated procedures for detecting mandatory features, feature groups, and implies/excludes relationships. Czarnecki *et al.* [155] proposed an algorithm based on binary decision diagrams for synthesizing feature models from logical formulae, thus leveraging formal methods to automate the feature modeling building process.

2.3.3 Variability Implementation Techniques

The theoretical foundations of SPLE must be complemented by practical mechanisms for implementing variability in software systems. Multiple approaches have been developed to realize variability, each with distinct trade-offs regarding flexibility, performance, and maintainability.

Preprocessor-based Approaches. Conditional compilation using preprocessors represents one of the earliest approaches to implementing variability [168, 169]. The Linux kernel extensively uses preprocessor directives for feature selection, enabling fine-grained variability management across millions of lines of code [170]. The historical study by Kuiter *et al.* [171] covering two decades confirms this approach, documenting the substantial growth in configuration complexity and demonstrating how the Kconfig system acts as a feature model that directly drives the use of preprocessor directives to adapt the kernel to extremely varied targets. However, preprocessor-based approaches can lead to code complexity and maintainability challenges, as demonstrated by Liebig *et al.* [172] in their analysis of preprocessor usage patterns.

Object-Oriented and Component-Based Variability. Object-oriented design patterns provide natural mechanisms for implementing variability through inheritance, polymorphism, and composition [173]. Gomaa [174] presents systematic approaches for designing variable architectures using object-oriented principles. Component-based software engineering offers additional flexibility by enabling dynamic composition and reconfiguration of system variants [175].

Aspect-Oriented and Feature-Oriented Programming. Aspect-oriented programming (AOP) enables crosscutting variability implementation by separating concerns that span multiple modules [176]. Mezini and Ostermann [177] demonstrate how aspects can be used to implement feature-based variability. Feature-Oriented Programming (FOP) provides direct support for feature composition, as shown by Batory *et al.* [178] and implemented in tools like AHEAD and FeatureHouse [179].

Runtime Variability and Dynamic Configuration. Runtime variability enables system adaptation without recompilation, providing flexibility at the cost of potential performance overhead [180]. Cetina *et al.* [181] explore autonomic com-

puting approaches for runtime feature activation and deactivation. This approach is particularly relevant for systems requiring dynamic adaptation to changing environmental conditions or user preferences.

In the context of browser configurations, Swanson *et al.* [182] proposed the `REFRACT` framework to avoid failures. Their approach involves adding guards to monitor each configuration and creating valid configurations when failures occur. While their work focuses on configuration testing, it highlights the importance of managing browser variability and represents a crucial tool for evolving software systems, where configuration changes may lead to operational errors. Beyond the software engineering domain, the application of feature modeling has been explored in other fields. Cashman *et al.* [183] proposed importing the software product line approach to the `Biobrick` repository in the biological domain, facilitating the planning of DNA element reuse. This demonstrates the potential of applying feature modeling techniques to diverse problem domains that involve managing variability, similar to the challenges faced in browser fingerprinting.

2.3.4 Applications to browser domain

The application of SPLE concepts to web browsers represents an emerging research area, given the inherent variability present in browser configurations and the potential for systematic variability management.

SPLE Application to Browser Fingerprinting. Modern web browsers contain numerous configuration options including flags, switches, settings, and optional components that can affect system behavior [29]. The Tor Browser Project exemplifies systematic configuration management for privacy, demonstrating how browser settings can be systematically controlled to reduce fingerprinting [184]. Browser fingerprinting research has documented how configuration differences create observable variations in browser behavior, suggesting underlying variability that could benefit from systematic management approaches [33].

Browser Extension Ecosystems as Product Lines. Browser extension systems exhibit natural product line characteristics, where a base browser platform supports numerous extensions that modify behavior and functionality. Kapravelos *et al.* [185] show how browser extensions create security vulnerabilities through unexpected feature interactions, highlighting the need for systematic variability management in browser ecosystems.

Cross-Browser Compatibility and Standards. The diversity of browser implementations creates a natural software product line scenario where web applications must adapt to different browser capabilities and behaviors. Mesbah *et al.* [186] propose automated approaches for cross-browser compatibility testing that leverage variability analysis techniques. Choudhary *et al.* [187] develop systematic approaches for detecting cross-browser inconsistencies that could benefit from SPLE-based analysis.

Configuration Space Analysis. The complexity of browser configuration spaces presents challenges similar to those addressed by traditional SPLE approaches in other domains. Browser configurations involve interdependent settings where changes to one parameter can affect the behavior of related features, creating con-

straint relationships analogous to those modeled in feature models [146]. Understanding these relationships is crucial for both browser developers and security researchers analyzing fingerprint resistance.

Performance Optimization Through Configuration Management. Browser performance depends heavily on configuration choices, creating optimization opportunities through systematic variability management. Butkiewicz *et al.* [188] characterize web page complexity and its impact on browser performance, suggesting that systematic configuration management could improve performance across different usage scenarios. This aligns with SPLE approaches to performance optimization through feature selection [189].

Privacy-Focused Configuration Management. The systematic management of browser configurations for privacy preservation represents a practical application of SPLE principles. The Tor Browser's approach to configuration standardization demonstrates how reducing configuration variability can enhance privacy by making users less distinguishable [184]. This exemplifies how SPLE concepts of managing variability can be applied to privacy engineering in web browsers.

The intersection of SPLE and browser engineering remains an active area of research, with initial work demonstrating the applicability of feature modeling and variability management techniques to browser configuration challenges. As browser complexity continues to increase, systematic approaches to managing configuration variability become increasingly relevant for both functionality and security considerations.

2.3.5 Challenges and Future Directions

Despite significant advances in SPLE theory and practice, several challenges remain that limit broader adoption and effectiveness of software product line approaches.

Scalability and Complexity Management. As product lines grow in size and complexity, traditional analysis techniques face scalability limitations. Benavides *et al.* [153] identify computational complexity challenges in feature model analysis that become prohibitive for large-scale systems. Apel *et al.* [190] explain that the number of products in a software line can grow exponentially with the number of features, making exhaustive analysis of interactions between features impractical. This combinatorial explosion motivates the search for advanced analysis strategies, whether exact and variability-aware or based on approximations and sampling.

Evolution and Maintenance Challenges. Long-term evolution of software product lines presents unique challenges not present in single-system development. Neves *et al.* [191] investigate safe evolution of software product lines, showing that changes to core assets can have far-reaching consequences across product variants. Seidl *et al.* [192] propose integrated approaches to managing evolution across multiple abstraction levels in software product lines.

Empirical Validation and Industrial Transfer. While SPLE shows theoretical promise, empirical validation of benefits remains limited. Berger *et al.* [193] survey industrial adoption of variability modeling, revealing gaps between research achievements and practical deployment. Clements *et al.* [144] identify fundamental organizational and technical barriers to SPLE adoption in industrial contexts,

emphasizing the need for systematic approaches to address adoption challenges. Schmid *et al.* [194] demonstrate that distributed development environments introduce additional complexity barriers that must be considered for successful SPLE implementation. Furthermore, Laguna *et al.* [195] highlight the challenges of evolving legacy systems toward product line architectures, revealing systematic gaps between theoretical frameworks and practical reengineering efforts.

2.4 Background Conclusion

This background review reveals a complex landscape where traditional user tracking mechanisms face increasing limitations, browser fingerprinting has emerged as a sophisticated identification technique, and software product line engineering offers systematic approaches to managing variability. However, several critical gaps remain that limit our understanding and ability to address browser fingerprinting challenges effectively.

Current browser fingerprinting research primarily focuses on identifying discriminating attributes and developing defense mechanisms, yet the systematic exploration of how browser configurations impact fingerprint generation remains largely unexplored. While studies have identified thousands of potentially fingerprintable attributes in the Browser Object Model, the relationship between specific browser configuration parameters and their impact on exposed attributes has not been systematically analyzed. This gap is particularly concerning given that users increasingly customize their browsing experience through settings, switches, flags, and extensions, potentially creating unique fingerprinting signatures without awareness of the privacy implications.

Furthermore, existing approaches to representing and analyzing browser fingerprint datasets rely on ad-hoc formats that fail to capture the complex relationships and constraints between different attributes. Traditional implementations of JSON-based representations in browser fingerprinting research do not adequately model the hierarchical nature of browser configurations or explicitly capture the dependencies between different fingerprinting attributes. This limitation hinders comprehensive analysis of fingerprint variability and makes it challenging to develop systematic approaches for privacy protection.

Software Product Line Engineering provides established methodologies for managing variability in complex software systems, yet its application to browser fingerprinting remains underexplored. Feature modeling techniques, successfully applied to manage variability in various domains, offer promising approaches for representing and analyzing browser fingerprint variability in a structured manner. However, the adaptation of these techniques to the specific challenges of browser fingerprinting requires novel approaches and empirical validation.

These identified gaps motivate the need for systematic approaches to: (1) comprehensively analyze the impact of browser configuration parameters on fingerprint generation, and (2) develop structured representations that capture fingerprint variability and enable sophisticated analysis of browser configurations. The following chapters address these challenges through large-scale empirical analysis and formal modeling approaches that advance our understanding of browser fingerprinting.

Chapter 3

Large-Scale Analysis of Configuration Impact on Browser Fingerprints

Chapter Contents

This chapter introduces FP-Rainbow, a systematic approach to explore and analyze the impact of browser configurations on browser fingerprints and the Browser Object Model (BOM). We collected 61,559 fingerprints from 18 Chromium versions, in both headful and headless modes, by testing 1,748 configuration parameters (switches). Our results show that 32 to 56 of these parameters have a detectable impact on the BOM. We document the attributes affected by each parameter and propose a method to re-identify configurations from fingerprints, achieving an average success rate of 84.36%. Finally, we demonstrate that a dataset collected on a single device can be effectively used for identification across multiple devices.

This chapter combines material from the contribution [1] *M. Huyghe, W. Rudametkin, and C. Quinton, "FP-Rainbow: Fingerprint-based browser configuration identification," in Proceedings of the ACM on Web Conference 2025, pp. 4325–4335, 2025.*

3.1 Introduction

Web browsers, now used by billions of people, constantly evolve to meet the Internet's growth and user expectations. They incorporate new features, allowing developers to deploy more sophisticated apps with each version. Simultaneously, users can personalize their browsing experience through the various settings of their browsers. A Europe-wide study shows that 36% of users reported having modified their browser settings to prevent or limit the use of cookies.[1] Combined with the popularity of many browser extensions, many users personalize their browsing experience. This, however, leads to browser configurations that are potentially detectable due to side effects. Some configurations may change the performance or behavior of a browser, while others may affect the *Browser Object Model* (BOM) and JavaScript APIs exposed by the browser, making them thus detectable. The BOM is distinct from the *Document Object Model* (DOM). While the DOM focuses on the content of a web page, the BOM is a programming interface that enables interaction with the browser. It provides access to objects, attributes, properties, and methods associated with the browser window. This makes it possible to retrieve more information about the browser's environment and configuration. These side-effects leak information to attackers that can be used to identify exploitable configurations (e.g., unsafe or experimental settings) or to simply increase the uniqueness of browser fingerprints and improve tracking algorithms.

Browser fingerprinting is a well-known technique to re-identify browsers [30, 33, 46]. Most fingerprinting algorithms use a few dozen attributes to create a unique identifier. These attributes are chosen because of their entropy, which provides uniqueness, and their stability between executions. Common attributes include the user agent, language, screen resolution, time zone, and plugins, as well as more complex attributes, such as canvas images to fingerprint the graphics layers [114, 196], font enumeration to recover installed fonts [34, 112], browser extension fingerprinting [115, 123], and GPU [118] or CPU fingerprinting [119, 121]. However, it has been

[1] https://ec.europa.eu/eurostat/web/products-eurostat-news/-/edn-20220208-1

shown that the browser exposes a much larger set of attributes that can potentially be exploited by attackers [37]. By enumerating the *Browser Object Model* (BOM), starting from the `window` object, we find anywhere from 12 to over 16 thousand objects, attributes, and methods exposed. Furthermore, little is known about how a browser's configuration affects the BOM, nor about the side-effects browser developers introduce when developing new features. To the best of our knowledge, no work has explored the impact of browser configuration settings on browser fingerprints. This information is useful in different scenarios, for both developers but also to attackers. For example, developers need information about the client's environment and configuration parameters to better reproduce their environment and quickly fix bugs. For attackers, they can identify configurations that are known to have security issues and exploit them.

In this chapter, we present FP-Rainbow, an approach to systematically explore and identify browser settings and configurations that affect the *Browser Object Model* (BOM) and the browser's fingerprint. For every configuration parameter available, we execute the browser in both headless and headful modes.[2] FP-Rainbow enumerates its BOM and collects an extensive fingerprint of around 15 thousand attributes depending on the browser's version and configuration. We then analyze the fingerprints to identify which configuration parameters impact which fingerprint attributes, if any. With this information, we provide a method to reidentify configuration parameters from the browser's fingerprint.

The key contributions of our work are:

- A dataset consisting of an exploration of 1,748 configuration parameters in 18 versions of Chromium, for which we collected 61,559 browser fingerprints, each containing between 12,334 and 16,473 attributes.
- The identification of 32 to 56 configuration parameters, depending on the browser's version, that impact the BOM. We also document the attributes that are added, removed or changed for each configuration parameter across different browser versions, finding relatively few collisions between switches in terms of impacted attributes.
- Through 1,116 randomly sampled browser configurations, we show that a dataset calculated on a single device can be leveraged to successfully identify configuration parameters from other devices.
- A method for (re-)identifying browser configuration parameters from a browser's fingerprint by comparing attribute subsets, achieving an average successful identification rate of 84.36% across all browser versions and configuration parameters that impact the BOM.

The remainder of this chapter is structured as follows. Section **??** provides an overview of the current state of the art in the field of browser fingerprinting and motivates our work. Section 3.2 introduces FP-Rainbow. Section 3.3 outlines our experimental methodology, while Section 3.4 presents the results of our experiments.

[2] Unlike the headful mode, headless mode has no graphical user interface.

3.2 FP-Rainbow

We present FP-Rainbow, our approach to systematically explore browser configurations to identify the impact they have on the Browser Object Model (BOM) through the collection of exhaustive fingerprints. Our objective is to identify how each configuration parameter impacts the BOM and thus reveals information about the browser's configuration.

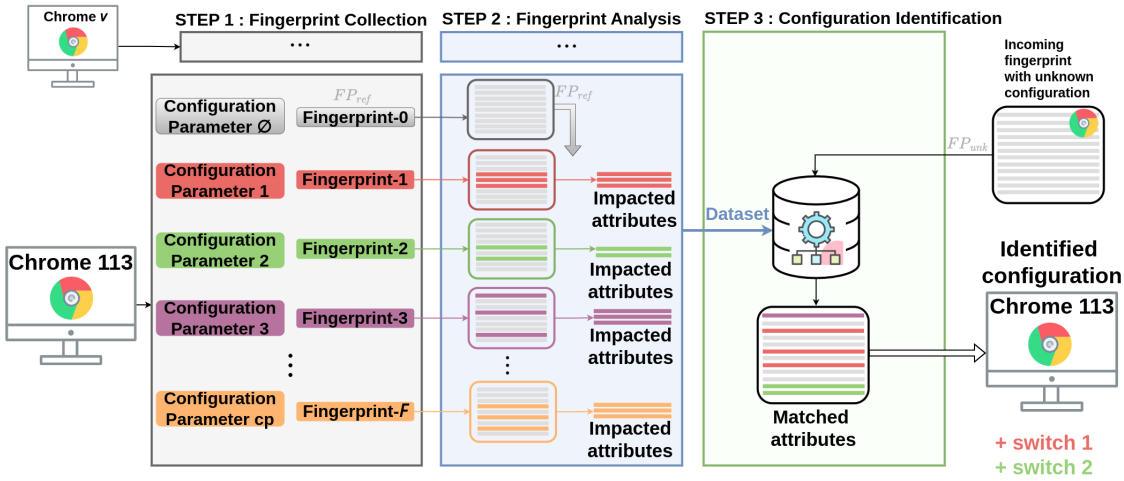


Figure 3.1: Overview of the FP-Rainbow approach

FP-Rainbow relies on a three-step process, as depicted by Figure 3.1. First, fingerprints are collected from different versions of Chromium. More precisely, one fingerprint is generated for each configuration parameter of each browser version. Second, each fingerprint is compared to the default browser configuration to determine which attributes are impacted by the configuration parameter. Many configuration parameters do not impact the BOM. We are interested in those that do. We use the knowledge of which configuration parameters impact what attributes of the BOM for version of the browser to:

- Given a fingerprint from a browser with an unknown configuration, we identify as many configuration parameters as possible that the browser has enabled.
- Calculate which attributes are good indicators of the activation of a configuration parameter by weighting their importance using a stability metric we develop.

In our current implementation, FP-Rainbow targets one type of configuration parameter, namely command-line switches,[3] but is extensible to flags and other configuration options.

3.2.1 Fingerprint Collection and Analysis

Fingerprint Collection. FP-Rainbow collects fingerprints and the browser's configuration, including the operating system it runs on, browser's version, and var-

[3] https://peter.sh/feed/chromium-command-line-switches/

ious configuration parameters. Specifically, FP-Rainbow systematically executes multiple instances of the browser, each with distinct versions and configuration parameters, and retrieves the resulting fingerprint. Such a fingerprint and its related browser configuration are then stored in a database to be further analyzed (see next steps). To delve into the details of the BOM, FP-Rainbow builds upon and extends two existing techniques, namely FingerprintJS[4], a popular open-source browser fingerprinting tool specialized in re-identifying browser's for tracking or authentication services, and JavaScript Template Attacks[5], a BOM enumeration solution developed by Schwarz *et al.* [37].

Fingerprint Analysis. Each generated fingerprint FP_{new} is compared with the reference fingerprint FP_{ref}, *i.e.*, the fingerprint devoid of any manually set configuration parameters, retaining only the default configuration parameters. This comparison aims to discern any distinctions introduced by the adjustment of configuration parameters. During the comparison, FP-Rainbow focuses on attribute differences between each FP_{new} and FP_{ref}. In particular, for every attribute in the fingerprint FP_{new}, FP-Rainbow distinguishes between four cases:

- The attribute exists in FP_{ref} with the same value.
- The attribute exists in FP_{ref}, but its value differs.
- The attribute does not exist in FP_{ref}.
- The attribute does not exist in FP_{new}.

FP-Rainbow associates each configuration parameter with the fingerprint it generates and identifies the impact on the browser's BOM. The objective is twofold: to help identify the attributes impacted by different configuration parameters and to build a knowledge base that can be leveraged to identify the configurations from unknown fingerprints. In short, we consider :

- FP_{ref}: The referent fingerprint which serves as the baseline for a specific browser version (one per version).
- FP_{new}: The fingerprint generated by applying a specific configuration parameter (one per parameter per version).
- FP_{unk}: The unknown fingerprint, where we aim to determine the configuration parameters used to generate it.

Configuration Identification. Relying on the dataset of impacted attributes, FP-Rainbow can attempts to identify configuration parameters from a browser by analyzing its fingerprint. To do so, for each configuration parameter from the unknown fingerprint FP_{unk}, attributes known to be impacted by a configuration parameter (from the previous fingerprint analysis step, *e.g.*, by a changed switch) are extracted and assessed, falling into one of the three following cases:

[4] https://github.com/fingerprintjs/fingerprintjs
[5] https://github.com/IAIK/jstemplate

- Added: The unknown fingerprint FP_{unk} has the attribute known to be impacted by the configuration parameter.
- Changed: The unknown fingerprint FP_{unk} has the attribute known to be impacted by the configuration parameter but with a different value.
- Removed: The unknown fingerprint FP_{unk} does not have the attribute known to be impacted by the configuration parameter.

Extracting subsets of attributes per configuration parameter. To identify an unknown fingerprint's configuration parameters, we compare it with the impact analysis for each configuration parameter from our dataset. The set of impacted attributes for each configuration parameter is small compared to the entirety of the fingerprint. We do not compare the fingerprints entirely since this will simply pollute the calculation, especially if multiple configuration parameters are activated or if the fingerprint originates from different environments. During the fingerprint analysis, as described previously, we identify the impact of each configuration parameter on the browser's fingerprint, in terms of attributes added, changed, or removed attributes. This provides us with a subset of attributes that are known to be impacted by the configuration parameter. We then extract the same attributes of the subset from the unknown fingerprint FP_{unk} to compare them. Similar to the analysis step, each attribute may exist, not exist, or exist with a different value. We then compare the extracted set of attributes from the unknown fingerprint FP_{unk} to those from the configuration parameter and decide if the parameter is active or not in the unknown fingerprint FP_{unk}. Given the amount of variability found in browser fingerprints, we cannot expect our comparisons to always be identical.

3.2.2 Stability Analysis of Impacted Attributes

During fingerprint generation, we collected between 12,334 and 16,473 attributes per fingerprint. Many attributes can vary between devices or across different browser versions. Additionally, a single configuration parameter may modify multiple attributes, or certain attributes might no longer be impacted in newer browser versions. To address this, we focus on identifying stable attributes that remain consistent in detecting configuration parameters. We introduce the concept of attribute stability and use it to quantitatively evaluate how stable specific attributes are across various browser versions under varying configuration parameters. An attribute is considered perfectly stable when it is influenced by only a single configuration parameter, regardless of the browser version.

Let CP be the set of configuration parameters, A the set of attributes, and V the set of browser versions. Then, cp is a configuration parameter from fingerprint FP_{new} such as $cp \in CP$. $cp_{default}$ represents the default configuration used in FP_{ref}. For each version $v \in V$, each attribute $a \in A$ is either associated with a version v and a configuration parameter cp, and is referred to as $a_{cp,v}$, or with a version v and no configuration parameter for the default configuration $cp_{default}$ and is defined as $a_{cp_{default},v}$. The difference between fingerprints FP_{ref} and FP_{new} for each attribute a is computed as follows:

$$Impacted_{a_{cp,v}} = 1 \text{ if and only if } a_{cp,v} \neq a_{cp_{default},v} \text{ (0 otherwise)} \tag{3.1}$$

Thus, $Impacted_{a_{cp,v}} = 1$ if the attribute a is impacted (changed, added, or removed). Else $Impacted_{a_{cp,v}} = 0$ if the attribute is not impacted. We evaluate stability across all configuration parameters $cp \in CP$. The more configuration parameters impact a given attribute, the more unstable the attribute is. If an attribute is impacted by only one configuration parameter, we consider it perfectly stable. To determine the stability of all attributes $a \in A$ for each configuration parameter $cp \in CP$, we check if an attribute a is impacted by all configuration parameters cp relying on the following formula:

$$Stability_{a,v} = (1 - \left(\frac{\sum_{cp \in CP_v} Impacted_{a_{cp,v}}}{|CP_v|}\right)) \tag{3.2}$$

Where $|CP_v|$ is the total number of configuration parameters that have impacted an attribute a for a version v. Thus, $Stability_{a,v} = 1$ if and only if the attribute a is impacted by only one configuration parameter $cp \in CP$. That is, $0 \leq Stability_{a,v} \leq 1$: the more configuration parameters impact a given attribute, the lower *Stability* while a value close to 1 signifies stability for attribute a across all configuration parameter $cp \in CP$ for a version $v \in V$.

To compute the stability of an attribute across all browser versions as well as to consider other environments or browser variations such as Headless and Headful, Equation (3.2) is iteratively applied over all versions $v \in V$. The result is then divided by the total number of browser versions $|V|$ as follows:

$$\omega_a = \frac{\sum_{v \in V} Stability_{a,v}}{|V|} \tag{3.3}$$

The overall stability of an attribute, ω_a, is thus computed such that $0 \leq \omega_a \leq 1$, with an attribute a being more stable as its related ω_a gets closer to 1.

Relying on Equation (3.3), one can measure the importance and stability of browser attributes across various versions and configuration parameters ($\approx 1,748$ switches). An attribute with a value ω_a close to 1 is highly important, stable, and reliable for use when identifying configuration parameters. Conversely, a value close to 0 indicates high variability and less reliability. This approach identifies essential attributes, minimizing time and resources for generating browser fingerprints while maintaining efficiency and accuracy in identifying configuration parameters.

3.2.3 Fingerprint Comparison for Configuration Identification

To identify a configuration parameter, we compare subsets of impacted attributes that we retrieved using $impacted_{a_{cp,v}}$ in equation 3.1. We define the set of impacted attributes resulting from our formula 3.1 as *impactedSet* for all versions, and $impactedSubset_v$ for a specific version, such that $impactedSubset_v \subseteq impactedSet$. During identification, we select only the attributes of the unknown fingerprint FP_{unk} that exist in our set *impactedSet*. By definition, the set of selected attributes from the unknown fingerprint FP_{unk} is less than or equal to the set of attributes in *impactedSet*. We have categorized the impacted attributes into three categories:

- $Add_{a,impactedSubset_v,unk} = 1$ if and only if the attribute a didn't exist in fingerprint $FP_{ref,v}$ but exists in $impactedSubset_v$ and in FP_{unk},

- $Chg_{a,impactedSubset_v,unk} = 1$ if and only if the value of attribute a is different in fingerprint $FP_{ref,v}$ and $impactedSubset_v$ but is identical between $impactedSubset_v$ and FP_{unk},

- $Rm_{a,impactedSubset_v,unk} = 1$ if and only if the attribute a exists in fingerprint $FP_{ref,v}$ but does not exist in $impactedSubset_v$ nor in FP_{unk}.

For each attribute, we have three cases, as indicated above (*i.e.*, Added, Changed, Removed). We count the occurrences of each case and normalize the value between 0 (the configuration parameter does not match) and 1 (the configuration parameter matches perfectly). To simplify we make the sum of the three case such as :

$$\begin{aligned} IdentifiedSet_v = & \, Add_{a,impactedSubset_v,unk} \\ & + Chg_{a,impactedSubset_v,unk} \\ & + Rm_{a,impactedSubset_v,unk} \end{aligned} \tag{3.4}$$

Then, we define:

$$Similarity_{(impactedSubset_v),(FP_{unk})} = \frac{\sum_{a \in A_{impactedSubset_v}} (IdentifiedSet_v)}{|A_{impactedSubset_v}|} \tag{3.5}$$

Where $|A_{impactedSubset_v}|$ is the number of impacted attributes a for a version v. From this equation, we can identify the configuration parameters cp used by an unknown web browser based on its fingerprint FP_{unk} by comparing each impacted attribute between the subset $impactedSubset_v$ and the unknown fingerprint.

The behavior of the $Chg_{a,impactedSubset_v,unk}$ operator is adapted for each type of data being processed, such as strings, numerical values, or PNG images. For numerical values, a simple absolute difference is often sufficient. However, to prevent large differences from skewing the comparisons, we assign normalize the differences. For strings, differences are measured using the Levenshtein distance. Regarding PNG images, particularly in the context of comparing Canvas fingerprints [114], the sum of the absolute pixel-by-pixel differences is generally sufficient. Given the large image size and the number of channels (*i.e.*, RGBA), this absolute difference is normalized similarly to numerical values. By summing the pixel-by-pixel differences, it reduces to a single numerical value, allowing PNG images to be processed and normalized as numerical parameters.

3.3 Methodology

This section describes our hardware settings, our data collection process, and the comparison methods we employed.

3.3.1 Research Questions

In this chapter, we propose an approach to systematically explore and identify browser settings and configurations that affect the BOM, and aim to answer the following research questions:

RQ1: *What impact do individual browser configurations parameters have on the BOM?* In particular, we want to assess the impact of one type of configuration parameters: switches.

RQ2: *Is it possible to identify a configuration parameter from an unknown fingerprint?* Comparing an unknown fingerprint to a set of known-to-be-impacted configuration parameters can provide insights into the unknown browser configuration.

RQ3: *Can a dataset generated from a single device effectively be leveraged to identify fingerprints across multiple devices?* If so, this could reduce the need for generating datasets on many, diverse devices, and instead, for example, allow calculating the dataset once on a more powerful server.

3.3.2 Software Setup and Hardware Settings

To isolate all components and enforce reproducibility, we conducted all our experiments using Docker and Docker Compose. Each software component, including the web browser, the database, the analysis tools, and the web server, is thus encapsulated in a Docker container, ensuring portability and reproducibility across different environments. Each container has its own Puppeteer[6] instance, and the same database is shared across all Docker containers. All experiments were conducted on a machine featuring an : 2x Intel(R) Xeon(R) Gold 5118 CPU and 188GB of RAM, operating on Rocky Linux with kernel version 4.18 for the dataset, and a laptop with an Intel(R) i7-1185G7 CPU and 64GB of RAM, operating on Arch Linux with kernel version 6.11, for the generation of unknown browser fingerprints.

3.3.3 Experimental Protocol

Fingerprint Collection.

We start Chromium browser with Puppeteer and inject configuration parameters into the browser's configuration. We run our browsers in a dedicated Docker container to facilitate reproduction and deployment. The configuration parameters are fetched from the database, a new entry is generated in the database with a UUID. The browser accesses our fingerprinting webserver that collects the fingerprint and cleans unstable attributes (for example, random unique identifiers and hashes that change at every execution). The database stores the browser version, browser configuration parameters, operating system, and CPU architecture, as well as the browser fingerprints. Browser fingerprints are stored in JSON files with unique identifiers (UUIDs) for quick retrieval. In case of a browser crash or unresponsiveness, the latest database entry is marked as `timeout`, this is a rare occurence. We launch one to four browser versions per thread, each with a distinct configuration. The collection

[6] https://pptr.dev/

is repeated for each configuration parameter $cp \in CP$ available in the database for each version of the browser. The initial launch for each version collects its default configuration, which serves as the reference fingerprint FP_{ref} during the subsequent analysis phase.

The fingerprint for each browser configuration is collected twice, in a row. The attributes that have been changed between the two fingerprints are discarded, stabilizing the fingerprint. Overall, $1,748$ switch configuration parameters have been successfully tested. Executing all configurations takes approximately 5 hours on our hardware configuration, as we test up to 4 browser versions per CPU thread. Once all fingerprints and their corresponding browser configurations are collected, the analysis can be started.

Fingerprint Analysis.

Certain attributes, including the test's URL, test timing, or network type, are not taken into account because they are specific to our test website or are unstable but not caught by our initial double fingerprint filter. Each remaining attribute of every fingerprint FP_{new} is compared with the reference fingerprint FP_{ref}, which uses the default configuration. When a difference is detected, the UUID of the fingerprint, the browser configuration, and the observed differences are stored (cf. Figure 3.2). By running the browser on the same platform, with the same configuration with the exception of one configuration parameter, we isolate the impact of the parameter. Throughout this step, we analyzed over $61,559$ fingerprints with 12 to 16 thousand attributes. After analyzing all versions, the resulting dataset captures the relationship between each configuration parameter and its impact on the BOM for each browser version. This dataset can then serve as a baseline to identify unknown fingerprints.

Configuration Identification.

To compare an unknown fingerprint FP_{unk} configuration parameters from the impacted attributes, we employed two strategies: one using the fingerprints generated from our BOM exploration and the other using FingerprintJS. In the former case, the fingerprint size can increase as the complete BOM is explored and each attribute-value pair is retrieved. As shown in Table 3.1, the number of attributes during our experimentation varied between $12,334$ and $16,473$. We selected the known impacted attributes from each switch (see Table 3.2) in our dataset and compare these attributes from the unknown fingerprint. As explained in Section 3.2, attributes are either `added`, `changed`, or have their values `removed`. Our identification process thus distinguishes between these three cases where we apply our Equation (3.5). In the case of the FingerprintJS library, a straightforward diff proved sufficient for all three cases given the fingerprint's relatively simple structure and the key/value pair similarity. These two approaches have allowed us to gain a deep understanding of the configuration parameters impact on the fingerprint. We have applied these techniques to all of our datasets.

3.4 Results

This section presents the results obtained through the utilization of our approach, FP-Rainbow. We first discuss the fingerprint collection and analysis phases. We then delve into the comparison phase and provide insights on configuration parameters that impacted an unknown fingerprint. All data gathered during our experiments, as well as additional data retrieved from other platforms (Windows and Debian) and hardware not reported in this chapter are are publicly accessible.[7,8]

3.4.1 Impact of Browser Configurations on the BOM [RQ1]

We generated fingerprints for 18 Chrome browser versions spanning over 18 months, for both the Headful and Headless variants, for a total of 36 browsers. The list of considered versions is available in Table 3.1. The list of switches that impact the BOM is provided in Table 3.2. This deliberate choice of a small sample of browser versions allowed us to exhaustively measure and comprehend the impact of each configuration parameter on the BOM.

As mentioned in the official Chromium documentation, the configuration for switches can be found on the *peter.sh* blog.[9] We tested a total of 1,748 switches. Throughout our tests, an average of 1,710 switches were successfully executed in the Headless browser version without encountering a timeout (3 timeouts were reported on each version) or a crash (23 to 144 crashed depending on version), and an average of 1,714 switches in the graphical version (22 to 38 crashes). Out of all launched switches, 12 to 15 switches affected the fingerprint from the FingerprintJS library, and 32 to 41 switches impacted the fingerprint from the BOM exploration in the headless Chrome version. In the Headful version, we observed 15 to 32 switches affecting the FingerprintJS, and 40 to 56 switches affecting the BOM exploration. The differences between each Headless or Headful version is due to the versions themselves, as we utilized all existing switches from the latest version, and some of these switches did not exist in earlier versions. However, the difference between Headless and Headful browser versions lies in the additional functions for graphical processing in the latter. This outcome opens the possibility of foreseeing a bot detection system designed for bots that operate without utilizing the graphical interface.

The majority of impacted fingerprints from FingerprintJS were also impacted in the BOM exploration, except for the switches that affected the canvas generation. Switches like `disable-readingfrom-canvas`, `force-prefers-reduced-motion`, and `force-high-contrast` were detected with the FingerprintJS library. The switch `disable-font-subpixel-positioning` was detected in the Headless Chromium version only, showing that certain configuration parameters also influence the canvas and can therefore be detected. The switch `headless` applied in Headful version shows that it impacts the browser fingerprint, and is therefore detectable on all the versions we tested. Some attributes can be affected by different switches, such as those ending with `length`, which are regularly impacted. The most affected attribute

[7] https://github.com/HuygheMaxime/fp-rainbow
[8] https://doi.org/10.5281/zenodo.13933676
[9] https://peter.sh/experiments/chromium-command-line-switches/

Table 3.1: Analysis of the effects of switches on the BOM, including the generation of browser fingerprints and the number of attribute for specific browser versions

Browser version	Number of switches that impact the BOM	Number of switches that impact FingerprintJS	Total Number of switches impacted	Fingerprint Generated	Min attributes per Fingerprint	Max Attributes per Fingerprint
Chrome-113.0.5672.63	45	15	50	1713	12894	15888
Chrome-114.0.5735.133	46	15	50	1713	12910	15917
Chrome-115.0.5790.170	46	15	50	1713	13014	16018
Chrome-116.0.5845.96	45	15	49	1713	13036	16029
Chrome-117.0.5938.149	45	17	49	1712	13062	16031
Chrome-118.0.5993.70	44	18	48	1712	13072	16080
Chrome-119.0.6045.105	44	18	48	1712	13070	16101
Chrome-120.0.6099.109	42	18	47	1712	13100	15945
Chrome-121.0.6167.184	42	17	46	1712	13120	15984
Chrome-122.0.6261.128	41	17	46	1712	13119	16052
Chrome-123.0.6312.122	42	17	45	1711	13164	16128
Chrome-124.0.6367.207	41	17	46	1710	13189	16134
Chrome-125.0.6422.141	40	16	44	1710	13300	16180
Chrome-126.0.6478.182	40	16	44	1710	13316	16227
Chrome-127.0.6533.119	40	16	44	1710	13335	16253
Chrome-128.0.6613.137	53	29	56	1722	13308	16279
Chrome-129.0.6668.58	54	30	58	1724	13310	16422
Chrome-130.0.6710.0	**56**	**32**	**61**	**1726**	13332	**16473**
HeadlessChrome-113.0.5672.63	37	12	40	1725	**12334**	15380
HeadlessChrome-114.0.5735.133	37	12	42	1725	12396	15409
HeadlessChrome-115.0.5790.170	37	12	40	1725	12500	15510
HeadlessChrome-116.0.5845.96	37	12	41	1725	12522	15521
HeadlessChrome-117.0.5938.149	36	14	40	1725	12540	15518
HeadlessChrome-118.0.5993.70	36	15	41	1725	12550	15567
HeadlessChrome-119.0.6045.105	35	15	40	1724	12558	15589
HeadlessChrome-120.0.6099.109	35	15	40	1725	12588	15433
HeadlessChrome-121.0.6167.184	35	15	40	1725	12608	15472
HeadlessChrome-122.0.6261.128	34	15	39	1724	12607	15540
HeadlessChrome-123.0.6312.122	33	15	39	1725	12650	15616
HeadlessChrome-124.0.6367.207	32	15	37	1604	12676	15623
HeadlessChrome-125.0.6422.141	33	14	38	1604	12757	15669
HeadlessChrome-126.0.6478.182	33	14	37	1723	12779	15722
HeadlessChrome-127.0.6533.119	34	14	37	1722	12785	15748
HeadlessChrome-128.0.6613.137	41	15	46	1696	13308	16279
HeadlessChrome-129.0.6668.58	41	15	46	1696	13310	16422
HeadlessChrome-130.0.6710.0	41	15	47	1694	13332	**16473**

Table 3.2: List of impacted switches across 18 tested Chromium versions in headless and headful modes, with the *min*, *max*, and *average* number of impacted attributes (`Added`, `Changed`, `Removed`), and the number of impacted versions (`Headful`, `Headless`, and `Total`).

Switch Name	Min	Max	Average	Headful versions Impacted	Headless versions impacted	Total versions impacted
–auto-open-devtools-for-tabs	11	11	11	5	0	5
–deterministic-mode	1	1	1	0	15	15
–disable-3d-apis	2220	2269	2256.66	18	18	36
–disable-databases	2	2	2	6	15	21
–disable-field-trial-config	1	88	20.85	18	3	21
–disable-file-system	3	3	3	18	18	36
–disable-gpu-driver-bug-workarounds	61	61	61	6	6	12
–disable-javascript-harmony-shipping	2	143	86.78	18	18	36
–disable-local-storage	52	52	52	18	18	36
–disable-notifications	5	5	5	18	18	36
–disable-permissions-api	31	31	31	15	15	30
–disable-remote-playback-api	2	2	2	18	18	36
–disable-scroll-to-text-fragment	29	29	29	18	18	36
–disable-shared-workers	2	2	2	18	18	36
–disable-software-rasterizer	2220	2269	2256.66	18	0	18
–disable-speech-api	52	54	53.11	18	18	36
–disable-speech-synthesis-api	47	49	48.11	18	18	36
–disable-threaded-compositing	1	1	1	18	18	36
–disable-web-security	4	4	4	11	11	22
–disable-webgl	2220	2269	2256.66	18	18	36
–dom-automation	26	26	26	18	18	36
–enable-begin-frame-control	1	1	1	0	15	15
–enable-benchmarking	15	18	16.33	18	3	21
–enable-blink-test-features	743	1059	887.64	18	18	36
–enable-experimental-web-platform-features	566	936	710.8	18	18	36
–enable-experimental-webassembly-features	2	9	6.06	18	18	36
–enable-field-trial-config	54	54	54	0	1	1
–enable-gpu-benchmarking	64	80	70.78	18	18	36
–enable-low-end-device-mode	2	2	2	18	18	36
–enable-nacl	428	590	458.85	18	3	21
–enable-net-benchmarking	17	17	17	18	3	21
–enable-network-information-downlink-max	4	4	4	18	18	36
–enable-precise-memory-info	2	2	2	18	18	36
–enable-privacy-sandbox-ads-apis	6	71	13.65	4	13	17
–enable-skia-benchmarking	29	41	34	18	18	36
–enable-stats-collection-bindings	27	27	27	18	18	36
–enable-webgl-developer-extensions	15	15	15	18	18	36
–enable-webgl-draft-extensions	13	13	13	7	7	14
–enable-webgl-image-chromium	2269	2269	2269	8	8	16
–force-fieldtrials	6	71	20	7	0	7
–headless	8	8	8	3	0	3
–ignore-gpu-blocklist	59	71	65.67	18	0	18
–instant-process	121	121	121	18	3	21
–javascript-harmony	2	53	10.2	10	10	20
–kiosk	2	2	2	0	3	3
–lang	2	2	2	0	15	15
–mangle-localized-strings	7	7	7	18	3	21
–override-use-software-gl-for-tests	15	15	15	18	0	18
–ozone-override-screen-size	10	11	10.33	0	3	3
–pdf-renderer	53	53	53	3	3	6
–shared-array-buffer-unrestricted-access-allowed	2	2	2	18	18	36
–start-fullscreen	2	2	2	0	3	3
–start-maximized	2	2	2	0	3	3
–touch-events	19	19	19	18	18	36
–use-first-party-set	3	12	8.25	2	6	8
–use-gl	2220	2256	2246.8	10	10	20
–use-gpu-in-tests	59	71	65.67	18	0	18
–user-agent	2	2	2	18	16	34
–variations-server-url	1	88	20.85	18	3	21
–web-sql-access	2	2	2	6	0	6
–whitelisted-extension-id	8	8	8	1	0	1
–win-jumplist-action	8	8	8	1	0	1
–window-name	8	8	8	2	0	2
–window-position	8	8	8	3	0	3
–window-size	8	8	8	3	0	3
–window-workspace	8	8	8	3	0	3
–windows-mixed-reality	8	8	8	3	0	3
–winhttp-proxy-resolver	8	8	8	3	0	3
–with-cleanup-mode-logs	8	8	8	3	0	3
–wm-window-animations-disabled	8	8	8	3	0	3
–xr__compositing	8	8	8	3	0	3
–xsession__chooser	8	8	8	3	0	3
–yuy2	8	8	8	3	0	3
–zygote	8	8	8	3	0	3

was `window._length`, which was impacted by ≈ 22 different switches followed by attribute `window.wgl_length` impacted by ≈ 8 different switches.

As explained earlier, some attributes can be impacted by different switches. However, in the majority of cases, each switch impacts the value of the attribute similarly, except for some attributes like `length` ones for which values can vary substantially. The limited number of collisions and relative independence allows us to avoid exploring all possible combinations of switches, which would be unfeasible given the number of switches, and instead create our dataset by enabling switches linearly, one at a time.

3.4.2 Configuration Identification [RQ2]

After assessing the impact of each configuration parameter, FP-Rainbow leverages this information to identify an unknown fingerprint. We evaluate FP-Rainbow's effectiveness for *(i)* identifying the browser fingerprints impacted by the configuration parameters in the dataset and *(ii)* generating and identifying browser fingerprints with several switches activated at the same time.

Baseline identification

First, we attempt to identify switches from fingerprints that already exist in our dataset from the data collection and analysis processes. This provides a baseline to understand if a switch is identifiable. As shown in Table 3.3, FP-Rainbow successfully identified the majority of the switches. During the experimentation, we observed that some switches enable the same feature and have exactly the same impact on the BOM, such as `disable-3d-apis` and `disable-webgl`. After verifying the source code[10], we can confirm they are perfectly identical. We consider such switches as *equivalent* switches. The dataset also contains cases where a switch is a perfect subset of another. This includes `disable-speech-synthesis-api` being a part of `disable-speechapi`. In some cases, a switch like `disable-3d-apis` can have both an equivalent and a subset like `disable-gpu-driver-bug-workarounds`. Identifying subsets allows us to distinguish if the child switch is enabled and not the parent switch. Table 3.3 also reveals that `Chrome` versions `128.0.6613.137`, `129.0.6668.58` and `130.0.6710.0` exhibit a significantly higher number of impacted switches compared to earlier versions.

[10] https://source.chromium.org/chromium/chromium/src/+/main:content/browser/web_contents/web_contents_impl.cc;l=3000;drc=7fa0c25da15ae39bbd2fd720832ec4df4fee705a

Table 3.3: Analysis of the effects of switches on the BOM and switch identification. *Left* provides statistics on the generation of browser fingerprints, attribute density for specific browser versions, results of individual switch identification from the BOM, equivalent and subset switches [RQ1]. *Middle-right* shows the results of the multi-switch identification experiment [RQ2] on the same platform that collected the dataset (Server), and *Right* shows the multi-switch identification platform with fingerprints from a different platform [RQ3].

	Impact [RQ1] and Identification [RQ2] of switches from browser fingerprints					Multiswitch identification experiment Server [RQ2]				Multiswitch identification experiment Laptop [RQ3]			
Browsers version	Switches that impacted the BOM	Identified switches	Failed switch identification	Equivalent switches	Subset switches	Number of switches tested	Identified switches	Mistaken identification (False Positive)	Unidentified (False Negative)	Number of switches	Identified switches	Mistaken identification (False Positive)	Unidentified (False Negative)
Chrome-113.0.5672.63	45	89.13% (41)	5	9	6	461	75.9% (350)	4.8% (22)	24.1% (111)				
Chrome-114.0.5735.133	46	89.36% (42)	5	9	7	751	76.3% (573)	2.9% (22)	23.7% (178)				
Chrome-115.0.5790.170	46	89.36% (42)	5	9	7	751	77.0% (578)	3.1% (23)	23.0% (173)	136	84.6% (115)	8.1% (11)	15.4% (21)
Chrome-116.0.5845.96	45	89.13% (41)	5	9	6	742	76.8% (570)	3.5% (26)	23.2% (172)	110	87.3% (96)	10.0% (11)	12.7% (14)
Chrome-117.0.5938.149	45	89.13% (41)	5	9	8	742	79.5% (590)	3.2% (24)	20.5% (152)	183	80.9% (148)	5.5% (10)	19.1% (35)
Chrome-118.0.5993.70	44	88.89% (40)	5	9	7	720	79.2% (570)	3.5% (25)	20.8% (150)	182	79.1% (144)	6.6% (12)	20.9% (38)
Chrome-119.0.6045.105	44	88.89% (40)	5	9	6	719	80.7% (580)	2.9% (21)	19.3% (139)	719	80.3% (577)	3.2% (23)	19.7% (142)
Chrome-120.0.6099.109	42	88.37% (38)	5	8	5	687	82.0% (563)	3.2% (22)	18.0% (124)	688	81.8% (563)	3.2% (22)	18.2% (125)
Chrome-121.0.6167.184	42	88.37% (38)	5	8	5	689	82.6% (569)	3.6% (25)	17.4% (120)	612	82.5% (505)	3.9% (24)	17.5% (107)
Chrome-122.0.6261.128	41	88.10% (37)	5	8	5	677	81.8% (554)	3.4% (23)	18.2% (123)	677	81.8% (554)	3.4% (23)	18.2% (123)
Chrome-123.0.6312.122	42	88.10% (37)	6	8	7	676	80.8% (546)	3.8% (26)	19.2% (130)	634	80.4% (510)	4.1% (26)	19.6% (124)
Chrome-124.0.6367.207	41	87.80% (36)	4	8	6	657	80.8% (531)	3.2% (21)	19.2% (126)	615	80.7% (496)	3.7% (23)	19.3% (119)
Chrome-125.0.6422.141	40	87.50% (35)	6	8	6	656	80.5% (528)	4.3% (28)	19.5% (128)	656	80.5% (528)	4.3% (28)	19.5% (128)
Chrome-126.0.6478.182	40	87.50% (35)	6	8	6	655	80.9% (530)	4.1% (27)	19.1% (125)	655	80.9% (530)	4.1% (27)	19.1% (125)
Chrome-127.0.6533.119	40	87.50% (35)	6	8	6	656	80.5% (528)	0.9% (6)	19.5% (128)	656	80.5% (528)	0.9% (6)	19.5% (128)
Chrome-128.0.6613.137	53	67.92% (36)	18	8	6	849	63.4% (538)	2.5% (21)	36.6% (311)	849	63.4% (538)	2.5% (21)	36.6% (311)
Chrome-129.0.6668.58	54	66.67% (36)	19	8	5	868	62.3% (541)	4.6% (40)	37.7% (327)	868	62.3% (541)	4.6% (40)	37.7% (327)
Chrome-130.0.6710.0	56	64.29% (36)	21	8	6	911	59.5% (542)	2.9% (26)	40.5% (369)	911	59.5% (542)	2.9% (26)	40.5% (369)
HeadlessChrome-113.0.5672.63	37	86.84% (33)	5	6	6	103	79.6% (82)	7.8% (8)	20.4% (21)				
HeadlessChrome-114.0.5735.133	37	86.84% (33)	5	6	6	613	79.4% (487)	4.7% (29)	20.6% (126)				
HeadlessChrome-115.0.5790.170	37	86.84% (33)	5	6	6	613	80.3% (492)	4.1% (25)	19.7% (121)	614	77.9% (478)	6.7% (41)	22.1% (136)
HeadlessChrome-116.0.5845.96	37	86.84% (33)	5	6	6	611	80.5% (492)	4.1% (25)	19.5% (119)	611	77.7% (475)	7.0% (43)	22.3% (136)
HeadlessChrome-117.0.5938.149	36	86.49% (32)	5	6	6	590	80.3% (474)	4.1% (24)	19.7% (116)	591	77.8% (460)	9.5% (56)	22.2% (131)
HeadlessChrome-118.0.5993.70	36	83.78% (31)	6	6	5	591	79.7% (471)	6.6% (39)	20.3% (120)	591	76.6% (453)	12.2% (72)	23.4% (138)
HeadlessChrome-119.0.6045.105	35	86.11% (31)	5	6	4	591	79.0% (467)	4.1% (24)	21.0% (124)	591	77.3% (457)	12.2% (72)	22.7% (134)
HeadlessChrome-120.0.6099.109	35	86.11% (31)	5	6	4	588	81.6% (480)	4.9% (29)	18.4% (108)	588	79.4% (467)	12.9% (76)	20.6% (121)
HeadlessChrome-121.0.6167.184	35	86.11% (31)	5	6	4	590	81.5% (481)	4.1% (24)	18.5% (109)	591	79.4% (469)	12.5% (74)	20.6% (122)
HeadlessChrome-122.0.6261.128	34	85.71% (30)	5	6	4	559	81.4% (455)	4.8% (27)	18.6% (104)	559	78.5% (439)	13.6% (76)	21.5% (120)
HeadlessChrome-123.0.6312.122	33	85.29% (29)	5	6	5	547	80.3% (439)	4.4% (24)	19.7% (108)	547	77.9% (426)	13.5% (74)	22.1% (121)
HeadlessChrome-124.0.6367.207	32	87.88% (29)	4	6	5	529	82.4% (436)	6.0% (32)	17.6% (93)	529	79.4% (420)	14.2% (75)	20.6% (109)
HeadlessChrome-125.0.6422.141	33	88.24% (30)	4	6	5	550	82.5% (454)	4.2% (23)	17.5% (96)	550	78.2% (430)	13.3% (73)	21.8% (120)
HeadlessChrome-126.0.6478.182	33	85.29% (29)	5	6	5	549	80.1% (440)	3.8% (21)	19.9% (109)	549	74.7% (410)	10.6% (58)	25.3% (139)
HeadlessChrome-127.0.6533.119	34	85.71% (30)	5	6	5	558	79.4% (443)	4.3% (24)	20.6% (115)	558	76.2% (425)	10.9% (61)	23.8% (133)
HeadlessChrome-128.0.6613.137	41	80.49% (33)	9	5	6	656	76.1% (499)	4.7% (31)	23.9% (157)	656	75.0% (492)	5.5% (36)	25.0% (164)
HeadlessChrome-129.0.6668.58	41	78.05% (32)	9	5	4	656	74.7% (490)	6.7% (44)	25.3% (166)	656	74.7% (490)	6.6% (43)	25.3% (166)
HeadlessChrome-130.0.6710.0	41	78.05% (32)	9	5	4	657	74.1% (487)	7.2% (47)	25.9% (170)	657	74.1% (487)	7.0% (46)	25.9% (170)

Multi-switch identification

We repeated the same experiment described in Section 3.4.2 but with multiple switches. Since it is impractical to explore every possible combination and there is no statistical data on the number of switches users typically activate, we devised a random sampling strategy to obtain representative samples. We generated 31 fingerprints per browser version. To cover as many browser fingerprints as possible, we used a pool of switches, distributing them evenly across tests. Specifically, we generated 10 browser fingerprints using $\frac{1}{4}$ of the available switches, then repeated the process with $\frac{1}{2}$ of the switches and $\frac{3}{4}$ of the switches. Finally, we generated a fingerprint incorporating all the switches impacted by this version. The result was a successful identification rate of 78.15% across all fingerprints with multiple switches, including identifying various switch combinations across all tested versions, as shown in Table 3.3.

3.4.3 Leveraging Single-Device Dataset for Fingerprint Identification Across Multiple Devices [RQ3]

We replicate the experiment described in Section 3.4.2, that was run on a server, but this time collect fingerprints from a different environment using a laptop. The setup remains identical to that in Section 3.3.2. As shown in Table 3.3, some identification results are identical or very similar for fingerprints generated on the two different machines. However, the differences between the two experiments show that changing the environment impacts the BOM. On average, FP-Rainbow achieves a recognition rate of 77.54%, compared to 78.15%, when testing fingerprints from a different environment, showing we can leverage single-device datasets to a large extent. Despite technical issues with older versions (113 and 114) which are not shown in the Table, the results indicate that FP-Rainbow can successfully identify configuration parameters in environments other than the original dataset.

3.4.4 Discussion

Minimal Fingerprint. Using FP-Rainbow, it is possible to determine a minimal browser fingerprint for the identification of configuration parameters. To find out this minimal fingerprint, proper attributes must be selected. To do this, the *Stability* formula (see formula 3.2) is applied when aiming for one specific version, or the ω_a formula (see formula 3.3) to broaden the scope over all versions. Attributes are then ranked based on stability. If an attribute is consistently linked to a single switch across all versions, it is considered stable and ranked higher. Conversely, if an attribute is impacted by multiple switches and/or only on certain versions, it is considered unstable and ranked lower. A threshold is then set for selecting the number of attributes, either regarding performance or precision. Selecting more attributes increases identification accuracy, but also extends the time required for generating and processing the fingerprint. This approach allows developers to inform users of sub-optimal or inadequate configurations and suggest improvements when visiting the website.

Limitations. As previously highlighted, the attributes impacted by switches may sometimes overlap with another switch and reduce the identification rate, such as with the `disable-webgl` and `disable-3d-apis` switches (see Section 3.4.2). Additionally, we noticed some noise during our analysis, like unstable attributes, or attributes impacted outside of our configuration parameters. This noise can be mitigated through an in-depth analysis of each attribute, which may lead to the exclusion of such attributes during the identification phase or the assignation of a lesser weight due to their instability or randomness. Several factors also introduce potential threats to the validity of our study. First, our dataset was exclusively generated using Headless and Headful Chromium versions within a Docker environment, which may restrict its generalizability to real-world scenarios.

3.5 Summary

While this chapter demonstrates the significant impact of browser configurations on the Browser Object Model (BOM) and the feasibility of identifying these configurations from fingerprints, the inherent variability and complexity of browser fingerprints, even under controlled conditions, present ongoing challenges. The empirical analysis conducted here highlights the need for a more structured and formal approach to represent and manage this variability effectively. Building upon these findings, the subsequent chapter introduces a formal representation of browser fingerprints, aiming to tame their variability and provide a more robust framework for understanding their characteristics across diverse configurations and environments.

```
{
  "85a43f31-84b7-4647-b4b0-696ed8fc20ea": {
    "window.Object": [
      "function Object() { [native code] }"
    ],
    "window.Function": [
      "function Function() { [native code] }"
    ],
    "window.openDatabase": [
      "function openDatabase() { [native code] }"
    ],
    "window._length": [
      924
    ]
  }
}
```

```
{
  "b3c20d9f-e617-466c-8aae-d7c8bc9bb072": {
    "window.Object": [
      "function Object() { [native code] }"
    ],
    "window.Function": [
      "function Function() { [native code] }"
    ],
    "window._length": [
      923
    ]
  }
}
```

Figure 3.2: Impact of the switch `-disable-databases` on the BOM enumeration (excerpt) for the `HeadlessChrome/117.0.5938.149` version. When the `-disable-databases` switch is not activated (left), the attribute `window.openDatabase` is present and the attribute `window._length` has a value of 924. When this switch is activated (right), the attribute `window.openDatabase` disappears and the value of `window._length` changes from 924 to 923.

Chapter 4

Formalizing and Taming Browser Fingerprint Variability with Feature Models

Chapter Contents

This chapter builds upon the large-scale analysis of configuration impact on browser fingerprints discussed in the previous chapter. It introduces a novel approach for formally representing browser fingerprints using feature models to capture and manage their inherent variability. This structured representation provides deeper insights into fingerprint evolution, facilitates browser comparisons, and offers practical applications such as enhanced configuration identification, while also presenting promising avenues for synthetic fingerprint sampling.

This chapter combines materials from the contributions published in *28th ACM International Systems and Software Product Line Conference* [2] and *MADWeb*

2025-Workshop on Measurements, Attacks, and Defenses for the Web [3]. We use the dataset of BrowserFM [3] because it is more recent, has more browser fingerprints, and is generated from two different hardware platforms. The conclusion drawn from this results is identical to that of the contributions presented in this chapter.

4.1 Introduction

The browser is the primary interface through which we interact with online services and applications. Web browsers have become ubiquitous tools, deployed across a wide spectrum of applications and use cases, offering an extensive array of functionalities and maintaining compatibility with numerous devices and peripherals. However, the increasing complexity and variability of web browser configurations poses significant privacy and security challenges. Users can customize their browsing experience through various configuration options, including settings, flags, command line switches and a diverse ecosystem of extensions. This wealth of customization options is known to render browser instances distinctly unique and identifiable through the aggregation of distinct attributes used to create unique identifiers, commonly referred to as a browser fingerprint [29].

Web browsers expose information about the user's device [118, 119, 122], operating system, fonts [112], extensions [115, 123], and various other technical characteristics. Fingerprints are often unique because web browsers disclose diverse attributes of their computing environments, including hardware specifications, operating system parameters, installed fonts, browser extensions, among other technical characteristics. While this information, exposed mainly through JavaScript APIs and attributes, allows websites to enhance and customize the user experience, it also raises significant privacy concerns.

Browser fingerprint tracking can compromise user privacy through sophisticated tracking methods that persist even when users attempt to maintain anonymity through conventional means, such as private browsing modes or cookie deletion. The granularity of fingerprinting attributes—ranging from hardware specifications to software configurations—can uniquely identify users across different websites and sessions. Of particular concern is the high entropy of certain attributes, which makes them especially effective for tracking purposes. As web technologies evolve and new features are rapidly introduced, the variability of browser configurations increases, making it challenging to understand and manage the potential risks associated with the exposure of the browser's JavaScript attributes. This challenge is further complicated by the rapid evolution of browser technologies and the continuous emergence of new fingerprinting vectors, necessitating automated approaches for comprehensive identification and assessments.

Traditional approaches to studying browser configurations, such as storing them in JSON format [37, 129, 197, 198], do not adequately capture the complex relationships and constraints between different attributes nor their mappings to the browser's configuration. Developers and privacy advocates face the challenging task of identifying and mitigating these discriminating characteristics while maintaining existing browser features.

We argue that feature models, a compact tree-based representation of variabil-

ity in software, can provide developers with abstractions and tools to better reason about browser fingerprints and to better identify and understand side-effects that lead to "fingerprintable" privacy issues. Feature models allow representing in a structured way the links between features with their constraints. Because browser fingerprints reflect the software and hardware variability of a system and are constructed through designed APIs, we have found it straightforward to represent fingerprints as feature models, allowing for a more elegant, efficient and lossless representation of large browser fingerprint datasets.

We propose a novel approach that leverages feature modeling techniques to represent browser fingerprints in a structured and comprehensive manner, exploring the links between browser fingerprints and the underlying configurations (*i.e.*, browser, system and hardware) that make them unique. By translating each fingerprint into a tree structure, we aim to gain a deeper understanding of the variability and uniqueness of browser fingerprints across different browser configurations. Our approach suggests developers undertake a comprehensive exploration of browser configurations (*e.g.*, switches, flags, settings), capturing a unique browser fingerprint for each configuration variant. This meticulous collection allows for the detection of even the most subtle changes. This representation allows for a more fine-grained analysis of the relationships between browser attributes and their impact on fingerprint distinctiveness, enabling developers to identify links between attributes of a browser fingerprint and the hardware, software or browser configuration that affected it.

The main contributions of this chapter are as follows:

- We introduce the idea of representing the browser fingerprint variability using the well-known technique of feature modeling for software variability [154], providing an elegant framework for reasoning about browser fingerprints and their underlying configurations.

- We propose algorithms for building individual feature trees from configurations and merging them into a unified tree, creating a comprehensive representation of the browser configuration landscape that captures both fingerprints and their corresponding system configurations.

- We provide a dataset of $89,486$ browser fingerprints from Chromium. Our fingerprint dataset explores $1,748$ switches from 13 versions of both headless and headful Chromium, which we use to build a feature model with $35,857$ nodes, demonstrating the scalability of our approach with extensive system and hardware configuration information.

- We provide insights into browser fingerprinting, present a methodology for identifying a minimal set of attributes necessary to identify partial browser configurations from their browser fingerprints, and discuss the implications of our approach for developing privacy-enhancing tools and assisting developers.

The remainder of this chapter is structured as follows. Section 4.2 provides background information on web browser configurations and motivates the need for a structured representation. Section 4.3 explains how fingerprint variability is captured, and details the process of building and merging browser fingerprint feature

trees. Section 4.4 describes various use cases of our approach and discusses the insights gained from our analyses. The background and related work on browser variability and feature model synthesis has been covered in Chapter 2.

4.2 Background and Motivation

Web browsers have evolved into highly complex software systems serving billions of users globally. Their continuous evolution is driven by the need to accommodate emerging web technologies and user requirements. Browser vendors enhance functionality through both in-house development and the integration of third-party components [140]. This leads to increasingly sophisticated and complex codebases, resulting in an extensive ecosystem of browser implementations. This evolution has spawned a diverse landscape of browser variants, each offering distinct feature sets and capabilities. The proliferation extends beyond traditional desktop browsers to include specialized variants for mobile devices, ARM-based architectures, and various operating systems. Furthermore, end users can extensively customize their browsing environment through extensions, settings, experimental flags, and over a thousand switches.[1] These layers of complexity—from core browser implementations to user-level customizations—create unique browser configurations, where each configuration variant potentially influences the browser's fingerprint, creating distinctive signatures that reflect both the underlying browser architecture and user personalizations.

Modern Web browsers have evolved into highly-variable software systems. This variability arises from multiple dimensions: user personalization options (*e.g.*, extensions, settings, plugins), operating system compatibility (*e.g.*, Windows, macOS, Linux, Android, iOS), and diverse hardware platforms (*e.g.*, desktop computers, mobile devices, vehicles, IoT devices). This multi-dimensional variability creates an extensive space of configurations, which impacts browser fingerprint characteristics, creating a complex mapping between configuration parameters and fingerprint attributes. This complexity necessitates robust analytical tools to model and understand these configuration-fingerprint relationships.

Software Product Line Engineering provides an approach to managing variability in software systems [145]. In SPLE, feature models [154] are a commonly used tool to represent software variability. Feature models employ a hierarchical tree structure consisting of a root node and its descendant nodes. In this structure, each node may possess multiple children but is restricted to a single parent, with terminal nodes designated as leaves.

For a feature model to be complete, each feature must be associated with a hierarchical constraint. The most restrictive constraint is `mandatory`, which implies that the feature is always present in every configuration if the parent feature is also present. In contrast, the least restrictive constraint is `optional`, indicating that the feature may or may not be present if the parent feature is also present. In addition, group relationships can also be defined, such as `OR` and `XOR` relationships. In the former case, a parent feature includes one or more of its child features, while in the

[1] https://peter.sh/experiments/chromium-command-line-switches/

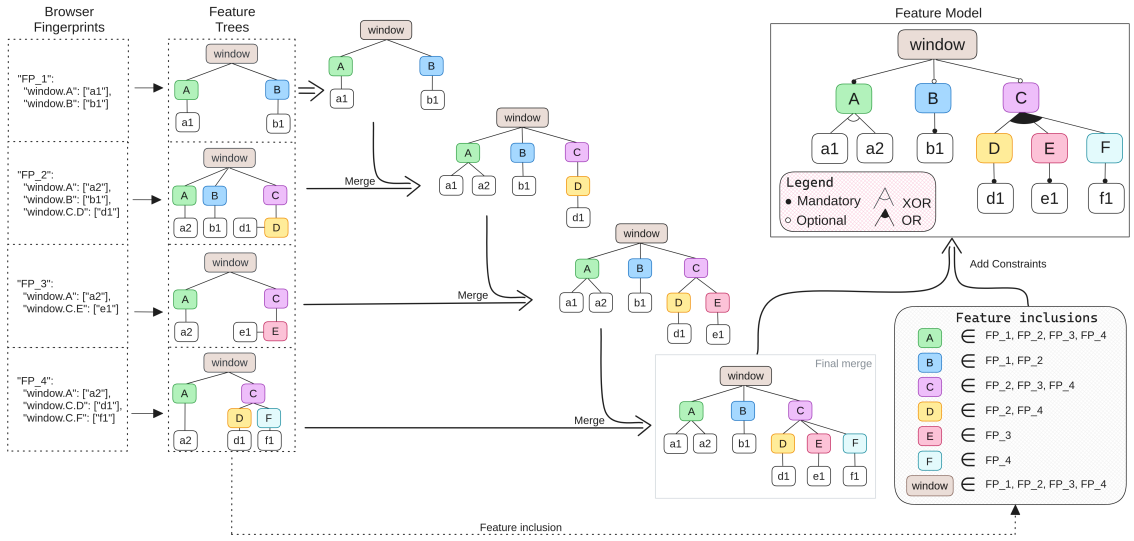


Figure 4.1: Feature Model Synthesis for Browser Fingerprints. We convert our dataset of fingerprints to feature trees. We record the list of configurations that contain a feature, which we call feature inclusions. Each feature tree is merged into a feature model. Mandatory, optional, XOR and OR constraints are calculated using parent-child relationships from the tree in conjunction with the list of feature inclusions.

latter, a parent feature includes exactly one of its child features.

4.3 Capturing Fingerprint Variability

To manage the complexity and changing nature of browser fingerprints, we propose an approach to synthesize a feature model that captures fingerprint variability.[2] Representing browser fingerprints as feature models provides a structured and systematic approach to understanding and analyzing the variability and uniqueness of browser fingerprints, as well as their practical implications. This approach is threefold, as depicted by Figure 4.1. First, one feature tree is built from each fingerprint. Then, one by one, each feature tree is merged. The final feature tree is then refined with parent-child constraints that are calculated using the list of feature inclusions extracted from the fingerprints.

The representation of browser fingerprints naturally maps to the hierarchical structure of feature models. This is arguably because the BOM exposes the browser's APIs, which are designed in a structured manner as any API is. We take each attribute from the fingerprint, seen in its fully qualified path form (*e.g.*, `window.screen.width`), and decompose it into a sequence of nodes called features in SPLE (*e.g.*, `window`, `screen`, `width`), beginning from the root node (*i.e.*, `window` for all browser fingerprints) and extending through each path component. In our model, attribute values are represented as leafs at the termination points of their respective branches.

[2] https://doi.org/10.5281/zenodo.11519529

4.3.1 Building Feature Trees

A browser fingerprint is a set of attributes with values (*i.e.*, key-value pairs). We build a tree from a fingerprint using the following rules:

- Each distinct attribute is transformed into an internal node in the tree structure;
- The corresponding values are represented as leaf nodes in the hierarchy;
- Nested attributes establish parent-child relationships, where containing attributes become parent nodes to their constituent attributes.

Figure 4.1 depicts the translation of 4 browser fingerprints into their respective feature trees. For instance, FP_2 consists of attributes `window.A` and `window.C.D` and two values, `a2` and `d1`. When applying the rules, a feature tree is created with `window` as the parent of both `A` and `C`, the latter being the parent of `D`, and two leaves, `a2` and `d1`. The same process is applied to each fingerprint, resulting in as many trees as fingerprints in the dataset. This structure maintains a one-to-one relationship between an attribute and its value within a single fingerprint, while allowing for the aggregation of different values across multiple fingerprints. Consequently, while an attribute path may be associated with multiple leaf nodes when considering different fingerprints, it maintains strict uniqueness within the context of any individual fingerprint. To build feature trees for the entire dataset ($89,486$ fingerprints), we parallelized the process using 24 threads, with each thread handling approximately $1,000$ fingerprints. This approach reduced the building time to around 2 hours and 20 minutes on a desktop computer equipped with an AMD Ryzen 9 7900X and 32GB of RAM.

4.3.2 Merging Trees

To build the tree that represents all the fingerprints in the dataset, each fingerprint tree is merged according to Algorithm 1. We start with one tree from the dataset (line 3) and, for every remaining tree (lines 4 – 14), nodes are merged into the tree (lines 5 – 9). The addition of each new node is performed by looking for its parent (using its name, as each attribute name is unique) and adding that node as a child of the parent node (lines 7 – 9). If no parent exists, the node is the root (line 11). In practice `window` is always the root. Merging trees built from the $89,486$ fingerprints results in a final tree with $35,857$ nodes. The merging process can be applied to any new feature tree, enabling the continuous evolution of the final tree over time. Unlike the case where all fingerprints would be merged into a single `JSON` file—resulting in the loss of their origins, this approach preserves the provenance of each fingerprint, eliminates redundancy, and provides a more structured storage solution.

4.3.3 Refining the Feature Model

Once all feature trees are merged, we apply a refinement process to identify the parent-child constraints (*i.e.*, mandatory, optional, alternative or exclusive features).

Algorithm 1 Merge trees

```
1: Input: trees                                   ▷ the set of feature trees
2: Output: mergedTree
3: mergedTree ← trees.get(0)
4: for tree ∈ trees\trees.get(0) do
5:     for node ∈ tree.nodes() do
6:         if ¬(mergedTree.contains(node)) then
7:             if node.hasParent() then
8:                 parentNode ← node.getParent()
9:                 mergedTree.get(parentNode).addChild(node)
10:            else
11:                mergedTree.setRoot(node)
12:            end if
13:        end if
14:    end for
15: end for
16: return mergedTree
```

We rely on the concept of *feature inclusion*, which is similar to *feature degree* defined by Metzger *et al.* [199]. We define a feature inclusion $FI(f)$, for a given feature f, as the set of configurations that contain f. More precisely, we compute the feature inclusion for every feature by recording the set of fingerprints $FP = \{FP_1, ..., FP_n\}$ that contain the feature, in order to determine the constraints (see bottom-right of 4.1). To address the limitations of constraints that do not establish relationships between different branches of the feature model, we assign a unique `UUID` to each generated fingerprint and associate it directly with each feature. This approach allows us to trace which attributes are influenced by specific configurations or environments, as each `UUID` is also stored in the nodes of the system information and browser information feature models. An additional advantage of this approach is the ability to reconstruct the original browser fingerprint from the feature model, even when the feature model contains a large collection of fingerprints. To identify mandatory and optional constraints, we use the following:

- ***Mandatory* constraint**. $FI(f) = FI(f_{parent})$, that is, when a node is present in all configurations where its parent node is also present, then it is marked as mandatory in regards to its parent in the feature model. For instance, feature `A` is always present when `window` is present and is thus mandatory in regards to `window`.

- All other features are marked *optional* and, if their parent node has more than one optional child, will be checked for the stricter *XOR* and *OR* constraints.

For features that have more than one *optional* child feature, we refine the parent-child relationship. Let us consider $OPT(f_{parent}) = \{opt_{child_1}, ..., opt_{child_n}\}$ the set of optional children of a parent feature f_{parent}. Then, for all fingerprints in FP and for all OPT sets in a fingerprint:

- ***XOR* constraint**. If in every configuration where the parent node exists, there is always one optional child node for that parent node, that is, $FI(f_{parent}) = \bigcup_{i \in OPT(f_{parent})} FI(F_{child_i})$, and the child nodes never overlap among themselves, that is, $\forall x, y \in OPT(f_{parent}), x \neq y : FI(x) \cap FI(y) = \emptyset$ then the children of that parent node are mutually exclusive, otherwise known as XOR.

- ***OR* constraint**. Similarly, if in every configuration where a parent node exists, there is an optional child node for that parent node, that is, $FI(f_{parent}) = \bigcup_{i \in OPT(f_{parent})} FI(F_{child_i})$, but there exists at least one child node that overlaps with another, that is, $\forall x, y \in OPT(f_{parent}), x \neq y : \exists FI(x) \cap FI(y) \neq \emptyset$, then the children of that parent node are in an *OR* relationship.

- ***Optional* constraint**. All remaining features are given the optional constraint. This means that, in general, $FI(f_{parent}) \neq \bigcup_{i \in OPT(f_{parent})} FI(F_{child_i})$, that is, there is at least one child that does not respect a strict *XOR* relationship or a weaker *OR* relationship.

4.3.4 Generating Browser Fingerprints and Linking Configurations

To construct our feature model,[3] we generate 89,486 browser fingerprints using the Chromium browser from versions *115.0.5790* to *127.0.6533*. We run each browser version thousands of times to collect a fingerprint for unique browser configurations. We also explore the browsers in both headless and headful configurations, an interesting use case to identify bots. During each launch, we record the browser's configuration, and importantly, its environment, which includes the system's hardware and software information.

By representing the browser fingerprint within the feature model, a framework traditionally used to represent configurations, we extend its application. Specifically, we create a new feature model based on the configurations and environments employed during browser fingerprint generation. We obtain three distinct feature models, each representing a different aspect of the process:

- The browser fingerprint feature model, encompassing 89,486 distinct fingerprints.

- The system information feature model, including the hardware and software characteristics of the system in which the browser was executed during fingerprint generation.

- The browser's configuration, detailing the browser version and its configuration parameters.

These three feature models enable us to determine which configurations impact the browser's fingerprint and the nature of their impact. Leveraging this understanding, we can identify the attributes in the browser fingerprint that are affected

[3] https://s.42l.fr/BrowserFM

by specific configurations and use this information to link a given fingerprint and re-identify its originating configuration parameter.

We constructed three feature models.[4] The first is based on 89,486 browser fingerprints and contains 35,857 nodes with 18,194 leafs (values). The second represents system information, with 464 nodes. The third captures browser information, comprising 92,998 nodes. These three feature models enable us to directly track configurations and their impacts on browser fingerprints, and provide different advantages and use cases.

4.4 Practical Applications

We have applied our feature model synthesis to a dataset of 89,486 browser fingerprints that we generated from 13 versions of the `Chromium` browser from 1,748 switches on Linux, in two variants (graphical and headless), for a total of 26 browsers. To generate the fingerprints, we systematically explored distinct configuration parameters. The fingerprint dataset, with each file stored in a JSON format, uses 121 GB of space. Interestingly, the feature model version uses 13.4 GB (*i.e.*, 9 times smaller) when storing feature inclusions (*i.e.*, the list of configurations that initially contained the feature). If we remove the feature inclusions, which are only necessary for calculating constraints, the size decreases to 2.1 MB, which is approximately 59,000 times smaller. The resulting feature trees range from 12,311 to 16,018 features, with a constant depth of 8.

This significant reduction enables long-term storage of a greater number of browser fingerprints, facilitating a more comprehensive understanding of their characteristics and evolution over time. The feature model can be leveraged to address various scenarios, such as random sampling for testing or debugging purposes, detecting anomalies, tracking the evolution of browser fingerprints, or identifying discriminating attributes.

4.4.1 Sampling Fingerprints

As discussed in Section 4.2, web application developers may benefit from randomly selecting and executing different browser configurations for application testing or for validating security and privacy functions. However, fingerprint datasets are often too small or not representative of user devices. Researchers often rely on small datasets [109, 124] or collaborate with popular websites to include their scripts to collect real browser fingerprints [45]. Projects such as Electronic Frontier Foundation (EFF)[5] or Am I Unique[6] are also dedicated to fingerprint collections, but the datasets from these sites are: *(i) not shared* for privacy concerns, *(ii) not exhaustive* because it is challenging to collect a set of configurations large-enough to cover variations in hardware, user settings, operating systems, browser versions, and other factors, *(iii) not representative* of average users since visitors to such sites are more likely to be aware of browser fingerprinting techniques and take privacy measures to counter

[4] `https://s.42l.fr/BrowserFM`

[5] https://coveryourtracks.eff.org/

[6] https://amiunique.org

them. A key challenge in browser fingerprint research lies in the limited access to real-world fingerprint data due to privacy concerns. Requests to share such datasets are often refused as they pose significant risks of exposing sensitive user information, limiting the ability to conduct extensive research or develop novel fingerprinting techniques. Additionally, privacy and user consent concerns can pose significant challenges in collecting and using browser fingerprint data.

Feature model-based sampling offers a promising solution to this issue [162]. By using the constraints and variability encoded in the feature model, one can generate synthetic browser fingerprints that are both realistic and diverse. That is, synthetic fingerprints are derived from a feature model based on an existing dataset. These synthetic fingerprints retain the structural characteristics and variability of real-world data without exposing the original, sensitive data. Moreover, even a small dataset of browser fingerprints can be significantly expanded through sampling techniques by systematically exploring the variability within the feature model and generating new fingerprints.

To tackle these issues, an alternative approach consists of generating fingerprints. However, fingerprint generation may result in invalid browser fingerprints [113]. By leveraging the synthesized feature model and applying existing sampling approaches [162], one can create a valid and relevant set of browser fingerprints to be used for testing purposes.

4.4.2 User Identification

BOM enumeration collects thousands of attributes per fingerprint, significantly increasing the overall fingerprint size compared to other more targeted approaches. The collection and comparison of browser fingerprints during each website visit presents substantial data management and storage challenges. While various techniques exist for user re-identification across multiple fingerprints,[7] including full fingerprint hashing and partial attribute subset hashing, these approaches have inherent limitations.

Although hashing provides efficient user re-identification, it proves fragile in practice, making it difficult to identify specific changes when hash values differ between visits. Hence, the selection of the attributes to create the hashed value must target those with high entropy that remain stable over extended periods of time. Furthermore, the redundancy of attributes across multiple browser fingerprints presents an additional challenge. The repetitive storage of identical attributes across multiple browser fingerprints introduces unnecessary computational overhead and storage inefficiencies, suggesting the need for an optimized data management strategy. These limitations in current approaches highlight the need for an innovative fingerprint representation that optimizes storage requirements while maintaining reliable identification capabilities and better comprehension of the browser fingerprint at a large scale.

[7] https://github.com/fingerprintjs/fingerprintjs

4.4.3 Fingerprint Evolution

Browser fingerprint generation always occurs client-side, while storage, analysis, and comparison are typically performed server-side. This is common practice because any client-side calculations can be spoofed and are thus better protected server-side. This architectural separation poses challenges for researchers, as it makes the fingerprinting process difficult to identify [200] and limits access to the fingerprint datasets and the evolution of fingerprints over time.

Previous efforts to address the challenge of understanding fingerprints overtime have relied on browser extensions to collect fingerprints from the same devices over long periods of time [43, 118]. While such solutions provide some utility, they also exhibit several limitations. For instance, extensions such as AmIUnique[8,9] provide users access only to their individual fingerprints but not to the entire dataset because it can be used to re-identify other users and may contain other sensitive data. Such an approach does not support the analysis of global fingerprint patterns or variations over time by the larger research community.

Furthermore, data collected via such tools is inherently biased, as users who install privacy-focused extensions typically exhibit higher privacy awareness and often utilize additional privacy-enhancing technologies, making them unrepresentative of the general population [45]. Finally, without access to high-traffic websites, it remains challenging to collect a substantial and representative dataset of browser fingerprints, especially one that spans over a long period. And in any case, browsers protect the device and limit what websites can collect. The browser's configuration, the system configuration, and the hardware's configuration and specifics cannot be collected through JavaScript, making any inferences difficult or limited because of the lack of ground truth values.

We propose an alternative to these website-based datasets, which contain information that may re-identify users and are thus not shareable nor otherwise reviewable by users or researchers. By exhaustively exploring browser configurations and collecting fingerprints on our own hardware, we can generate datasets that better understand the fingerprinting surface of browsers, the links between configuration parameters and fingerprinting attributes, and we can share the datasets freely since they no longer re-identify other users. The tradeoff is a loss of the extensive diversity of devices found in other datasets [29, 33, 45] for a gain in controlling and collecting the entire hardware, system and browser configurations.

Our approach relies on exhaustively exploring browser configurations (*i.e.*, browser version, switches, flags and settings) on hardware in our control for which we also collect extensive configuration information. Furthermore, by representing fingerprints as feature models, existing approaches on feature model and software product line evolution [201, 202] can thus be applied. For example, by constructing a feature model for each browser version, it becomes easy to compare how and when browser features are introduced or deprecated, track changes across different browser versions, and identify attributes with high entropy.

Figure 2.1a illustrates the attribute changes in browser fingerprints from `Chromium`

[8] https://addons.mozilla.org/firefox/addon/amiunique/
[9] https://chrome.google.com/webstore/detail/amiunique/pigjfndpomdldkmoaiiigpbncemhjeca

versions 115 to 127. In particular, version 125 looks like a turning point: attributes were removed regularly until this version, while added attributes increased from here. To observe fingerprint evolution, we first synthesize a feature model from the default browser version, *i.e.*, a browser configuration devoid of *switches* or modified *flags*, and we then compute the feature model differences between two of these versions [201]. Relying on the set of synthesized feature models, we can thus track removed features (*e.g.*, due to BOM deprecation), or measure the impact on user privacy when new features are added to the browser.

Our experiments show that the number of nodes increases over time. Between `Chromium` versions 115 and 127, *i.e.*, 13 releases over 13 months, we observed an increase of $1,333$ new nodes, from $28,926$ to $30,259$ nodes. By leveraging the feature model of each browser version and applying random sampling techniques, browser developers can assess the privacy impact of newly implemented features. This approach allows them to explore a wide range of configurations, identify potential privacy risks, and make informed decisions about the design and implementation of new functionality.

4.4.4 Browser Comparison

There is a wide variety of web browsers. However, the majority of popular browsers are currently derived from Chromium. Some browsers, such as *Brave*,[10] *Ungoogled Chromium*,[11] and *Epic Privacy Browser*,[12] place a strong emphasis on privacy. Two of these browsers specifically claim to protect users against browser fingerprinting through various techniques. By encoding browser fingerprints in feature models, we can facilitate comparisons between Chromium and its derivatives to then evaluate the effectiveness of their fingerprinting protection mechanisms. We can compare these browsers to gain a deeper understanding of how their differences, and more specifically, their anti-fingerprinting tools impact browser fingerprints.

4.4.5 Configuration Identification

To generate our fingerprint dataset, we exhaustively explored the browser's runtime configuration space, focusing specifically on command-line switches.[13] Switches[14] are command-line arguments initialized before the browser's launch. We explored $1,748$ switches. We observed that fingerprint feature nodes appear or disappear depending on the (de)activation of certain configuration parameters. These observations raised questions regarding whether these nodes could be identifiable and linked to the browser's configuration. To facilitate the identification of these configuration parameters, we computed differences between a feature tree resulting from the default configuration and feature trees generated from configurations where *switches* and *flags* were activated one by one. Computing these differences was performed

[10] https://brave.com
[11] https://github.com/ungoogled-software/ungoogled-chromium
[12] https://epicbrowser.com/
[13] https://chromium.googlesource.com/chromium/src/+/main/docs/configuration.md
[14] https://peter.sh/feed/chromium-command-line-switches/

relying on the method proposed in [201]. A *switch* or *flag* parameter was thus considered to impact the fingerprint when added or removed nodes resulted from that diff operator.

We stored a mapping for each *switch* and *flag* to their fingerprint. Some of these configuration parameters are highly identifiable because they enable or disable features in the web browser that impact the browser fingerprint and therefore our feature model. We also generated new browser fingerprints in more recent versions on different hardware and operating systems using *switches* that we knew were identifiable. Subsequently, we attempted to verify whether we could identify the *switch* used from the browser fingerprint. In the majority of cases, we successfully identified the *switch* from the browser fingerprint. In instances where we did not correctly identify it, the set of nodes was too close or identical to another case. With this information, we should be able to identify browser configurations from their fingerprints and potentially provide tips on how to make the configuration more privacy-friendly, if desired. Another use case is for developers. If a user reports a bug and generates a report, we can easily add the browser fingerprint to the report. This allows the developer to reproduce the user's configuration, gaining a better understanding of the bug, and correcting it more efficiently.

4.4.6 Reduction of Fingerprinting Detection

Recent research in anti-fingerprinting technologies has focused extensively on analyzing attribute sets targeted by fingerprinting scripts [34, 41, 200, 203]. Understanding the complex relationships between browser configurations and their corresponding attributes is crucial for developing sophisticated fingerprint algorithms as well as optimal spoofing countermeasures. This knowledge enables the implementation of dynamic attribute selection mechanisms that can vary with each browser session, effectively enhancing privacy protection. By identifying minimal subsets of attributes necessary for re-identification, we can strongly reduce the attributes needed as well as the time to fingerprint a device. Furthermore, we find that many attributes are either activated or deactivated in groups when a configuration parameter changes, making the attributes have similar importance in identifying the browser or the configuration parameter. For example, WebGL support adds about $2,000$ attributes simultaneously to the BOM. Instead of collecting all WebGL-related attributes, the system may verify the presence of WebGL support and authenticate the browser version, preventing unnecessary attribute collection while maintaining robust spoofing detection capabilities due to the many similar attributes that can be tested. Identifying these attributes and selecting minimal sets can make the fingerprinting process much more efficient. Also, by implementing randomized attribute selection strategies on attributes with similar levels of importance, we can create dynamically generated fingerprinting scripts that can mitigate client-side attribute spoofing in a moving target approach. This targeted approach significantly improves both the efficiency and the effectiveness in detecting spoofed browser configurations.

4.5 Summary

In this chapter, we introduced a novel approach to formally represent browser fingerprints using feature models, aiming to capture and manage the inherent variability arising from browser, system, and hardware configurations. Our methodology, encompassing the building and merging of feature trees, enables a structured, efficient, and lossless representation of large-scale fingerprint datasets. This formalization not only provides deeper insights into fingerprint evolution and allows for browser comparisons, but also offers practical applications such as synthetic fingerprint sampling, enhanced user identification, and a systematic way to link fingerprints back to their originating configurations.

Chapter 5

Tools, Datasets, and Platforms

In this section, we present the tools, the datasets and platforms used in our research. Each tool and dataset is described in detail below, all are open-sourced and reproducible. An overview of the different datasets generated is provided in Table 5.1.

5.1 Platforms

5.1.1 Experimental Platforms

Our experiments rely on three main platforms with different hardware and software configurations. Below, we detail the software and hardware characteristics of each platform.

Typhon server Typhon is a server based on a Dell PowerEdge R540 equipped with 2× Intel(R) Xeon(R) Gold 5118 CPUs @ 2.30GHz (24 cores each) and 188 GB RAM, running Rocky Linux 8.10. Each browser fingerprint is generated by running a Chromium browser inside a Docker container. Headful mode is enabled using a virtual display with Xvfb. This setup ensures consistent and controlled environments for browser fingerprint generation and provides enough computing power to run all Chromium instances in parallel.

MLM desktop The MLM (Monoswitch Large-scale Measurement) platform is a desktop computer equipped with a Ryzen 9 7900x CPU, 32 GB RAM, an MX500 2TB SSD, and an RTX 3090TI GPU, running Zorin OS (Debian-based). For browser fingerprint generation, Chromium instances are executed inside Docker containers, with a single switch enabled at a time (as on the Typhon server). Although this platform has a physical display, it is not used during browser fingerprint generation.

Laptop The Laptop has been used to mimic a real user, this is a HP EliteBook 840 G8 with a 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz (4 cores) and 64 GB RAM running on Arch Linux. It is used for experiments involving multiple switches enabled simultaneously (multiswitch configurations) with a physical display.

Unlike the containerized Typhon and MLM platforms the Laptop platform runs Chromium natively on the host OS. This allows for the evaluation of real-world, user-like configurations and hardware-specific effects. The Laptop platform is used for validating results obtained from virtualized environments (Typhon and MLM).

Across all platforms, browser automation and fingerprint collection are orchestrated using the FP-Rainbow pipeline.[1] The combination of containerized (Typhon, MLM) and physical (Laptop) platforms provides a comprehensive view of browser fingerprint variability across both controlled and realistic settings.

5.2 Source code and Reproducibility

5.2.1 Code repositories

The source code for our experiments is available in two open-source repositories:

- **FP-Rainbow** (https://github.com/HuygheMaxime/fp-rainbow): This repository contains the complete pipeline for large-scale browser fingerprint collection from the contributions presented in Section 3, including scripts for automating browser launches with the collection of fingerprints and linking them to their corresponding configurations.

- **FP2FM** (https://github.com/HuygheMaxime/fp2fm): This toolkit provides tools for transforming, merging, and analyzing feature models from browser fingerprint datasets used in the contributions presented in Section 4.

5.2.2 Dataset

The dataset associated with *FP-Rainbow: Fingerprint-Based Browser Configuration Identification*, available at https://zenodo.org/records/13933676, contains a large-scale collection of browser fingerprints gathered from multiple Chromium versions (from 113 to 130) and across different hardware configurations. The dataset includes fingerprints collected in both headless and headful modes, using several platforms such as Typhon (Docker), MLM (Monoswitch), and a laptop (Multiswitch). It provides detailed information on browser configurations. It includes a set of enabled or disabled switches, the resulting fingerprints as measured through a complete exploration of BOM and by FingerprintJS[2] a popular open-source library for fingerprinting. This dataset is designed to support research on identifying browser configurations based on their fingerprints.

The dataset from *Taming the Variability of Browser Fingerprints*, accessible at https://zenodo.org/records/11519529, offers a comprehensive set of browser fingerprints collected from Chromium browsers (versions 109 to 115) in both headless and headful modes. The data was gathered using the Typhon platform (Docker) and focuses on systematically varying browser switches and flags to analyze their impact

[1] https://github.com/HuygheMaxime/fp-rainbow
[2] https://github.com/fingerprintjs/fingerprintjs

on fingerprint variability. The dataset includes detailed records of the browser versions, the specific switches and flags used, the resulting fingerprints (using BOM enumeration and FingerprintJS), and the corresponding hardware and software configurations. It is intended to facilitate the study of how browser configuration changes affect fingerprint uniqueness and stability.

The dataset for *BrowserFM: A Feature Model-based Approach to Browser Fingerprint Analysis*, available at `https://s.42l.fr/BrowserFM`, contains an extensive set of browser fingerprints collected from Chromium browsers (versions 115 to 127) in both headless and headful modes. The data was obtained using Typhon and MLM platforms (both Docker-based), and is structured according to a feature model that captures the relationships between browser configuration options. The dataset includes the fingerprints themselves (extracted from the BOM), the configuration of browser switches, and the corresponding nodes in the feature model. This resource is particularly suited for research on modeling and analyzing the configuration space of browsers and its impact on fingerprinting.

Title	Web Browsers	Versions	Hardware Configurations	Technologies	Total fingerprints	Total switches	Total flags	Total nodes FM	Attributes	Size
Taming the Variability of Browser Fingerprints[3]	Chromium, Headless, Headful	109–115 (7)	Typhon (Docker)	BOM	22,773	1,519	596	34,557	12,311–16,018	29 GB
FP-Rainbow: Fingerprint-Based Browser Configuration Identification[4]	Chromium, Headless, Headful	113–130 (18)	Typhon (Docker 61,921), MLM (Monoswitch 61,901), Laptop (Multiswitch 1,822)	BOM, FingerprintJS	189,737	1,748	0	—	12,334–16,473	—
BrowserFM: A Feature Model-based Approach to Browser Fingerprint Analysis[5]	Chromium, Headless, Headful	115–127 (13)	Typhon (Docker 44,752), MLM (Docker 44,734)	BOM	89,486	1,748	0	35,857	—	121 GB

Table 5.1: Detailed comparison of browser fingerprinting datasets. Each row represents a different contribution, with columns indicating the browser versions used, hardware configurations, technologies employed, total fingerprints collected, number of switches and flags, total nodes in the feature model (FM), attribute ranges, and dataset sizes.

5.2.3 Reproduction instructions

Browser Fingerprint Generation

All browser fingerprinting datasets have been generated using the FP-Rainbow pipeline. The following steps reproduce the fingerprint collection process:

1. **Clone the repository:** https://github.com/HuygheMaxime/fp-rainbow
2. **Install requirements:** Ensure `docker`,[6] `docker-compose`,[7] `bash`,[8] `git`,[9] `npm`,[10] `wget`[11] and `zstandard`[12] are installed.
   Your user should be in the `docker` group (see repository README for details).
3. **Prepare the environment:**
   (a) Move to the preparation directory: `cd 1_Fingerprint_Collection/app/pre`
   (b) Run the setup script: `./prep.sh`

   This step creates necessary directories, installs dependencies, and starts the database and web server.
4. **Download Chromium executables:**
   (a) Download the archive containing all required Chromium browser executables:
   `wget https://zenodo.org/records/13933676/files/browsers.tar.zst`
   (b) Extract the archive to the appropriate directory:
   `tar -use-compress-program=unzstd -xvf browsers.tar.zst -C 1_Fingerprint_Collection/app/build/`
   (c) Ensure that all desired Chromium executables are now present in `./1_Fingerprint_Collection/app/build/browsers/`
5. **Run the fingerprint collection:**
   (a) Move to the *client* directory: `cd ../client`
   (b) To collect fingerprints for all browsers and switches, run:
   `./multiple-browser.sh`
   (c) To collect fingerprints for multi-switch experiments, run:
   `./multiple-browser-multiswitch.sh`

   These scripts will automatically explore browser command-line switches (in both headless and headful modes) for all Chromium versions present in `app/build/browsers/`, and collect fingerprints.

[6] https://www.docker.com/
[7] https://docs.docker.com/compose/
[8] https://www.gnu.org/software/bash/
[9] https://git-scm.com/
[10] https://www.npmjs.com/
[11] https://www.gnu.org/software/wget/
[12] https://facebook.github.io/zstd/

6. **Access and verify results:**

   (a) Fingerprint data is stored in `app/server/jsta/data` (from BOM enumeration) and `app/server/fpjs/data` (from FingerprintJS).

   (b) The database can be accessed at http://localhost:8032/?pgsql=postgres&username=user&db=db_fp using credentials from `app/database/.env`.

Feature model generation

Each browser fingerprint is transformed into a single feature tree. Each of these feature trees is merged into a single feature model. The generation of feature models from browser fingerprints is performed using the open-source `FP2FM` toolkit.[13] This toolkit enables the extraction, merge, and analysis of feature models from large-scale browser fingerprint datasets. The following steps reproduce the feature model construction process:

1. **Clone the repository and set up the environment:**

   (a) Clone the repository:
   ```
   git clone https://github.com/HuygheMaxime/fp2fm
   cd fp2fm
   ```

   (b) Create and activate a Python virtual environment:
   ```
   python3 -m venv venv
   source venv/bin/activate
   ```

   (c) Install the required Python packages:
   ```
   pip install -r requirements.txt
   ```

2. **Prepare the dataset:**

   (a) Unzip the dataset archive (e.g., `dataset.zip`) to obtain the following structure:
   ```
   dataset/
     FeatureModel/
     fingerprints/
   ```

   (b) Copy the `fingerprints` folder (containing the fingerprint in `json` files) into the `fp2fm` directory, replacing any existing `fingerprints` folder if necessary.

3. **Build feature trees from fingerprints:**

   - For single-threaded processing (suitable for small datasets):
     `python3 src/tree_builder_anytree.py fingerprints fp_feature_tree`
   - For large datasets, split the `fingerprints` folder into numbered subfolders (e.g., `1/`, `2/`, `3/`) for parallel processing. Then, in a `tmux` session, run:
     `bash src/runall.sh N`
     where `N` is the number of subfolders.

[13] https://github.com/HuygheMaxime/fp2fm

4. **Merge feature trees into a single feature model:**

   - After feature tree generation, merge all resulting `.pkl` files into a single feature model:
     `python3 src/merge_fm.py fingerprints merged_tree.pkl`

5. **Analyze and process the feature model (optional):**

   - Print the feature model structure:
     `python3 src/print_tree.py merged_tree.pkl`
   - Count the number of features and leaves:
     `python3 src/count_features.py merged_tree.pkl`
   - Remove UUID/origin metadata from the feature model:
     `python3 src/remove_origin.py merged_tree.pkl merged_tree_without_uuid.pkl`

The resulting feature model can be used to store and analyze browser fingerprints as described in Section 4.3.

Chapter 6

Conclusions and Perspectives

6.1 Summary of Contributions

This thesis addresses critical challenges in understanding and analyzing browser fingerprinting through systematic exploration of configuration impacts and modeling approaches. The research contributions span both theoretical frameworks and practical methodologies that advance our understanding of browser fingerprint variability and its implications for privacy and security.

6.1.1 FP-Rainbow: Fingerprint-Based Browser Configuration Identification

FP-Rainbow, detailed in Chapter 3, represents the first comprehensive systematic approach for analyzing how browser configuration parameters affect fingerprint characteristics. We developed an automated pipeline that systematically explored $1,748$ configuration parameters across 18 versions of Chromium browser. This exploration generated a dataset of $61,559$ browser fingerprints, representing an extensive systematic study of browser configuration impact. Our analysis identified 32 to 56 configuration parameters that significantly impact the BOM depending on the browser version. We demonstrated that configuration parameters can be reliably identified from unknown fingerprints with an average success rate of 78.15%. Our work showed that from browser fingerprints generated from a single device, it is possible to identify configurations on multiple hardware platforms independent of those used to generate the fingerprints. This work provides quantitative evidence of how browser configuration choices directly affect user privacy exposure.

6.1.2 Taming the Variability of Browser Fingerprints

We introduced a novel approach for representing browser fingerprint variability using feature models from SPLE, as detailed in Chapter 4. This modeling approach transforms flat attribute-value pairs into hierarchical tree structures that capture relationships and constraints between fingerprint attributes. Our methodology automatically constructs feature models from large-scale fingerprint datasets through

systematic tree building and merging algorithms. The approach achieves remarkable storage efficiency, reducing dataset size by up to 59,000 times compared to traditional JSON-based representations while preserving complete information. We demonstrated how feature models enable sophisticated analysis capabilities including evolution tracking. The representation in Feature Models facilitates scalable analysis of browser fingerprint datasets containing millions of fingerprints across multiple browser versions.

6.1.3 BrowserFM: A Feature Model-based Approach to Browser Fingerprint Analysis

BrowserFM, detailed in Chapter 4, introduces a concrete methodology for applying feature models from SPLE to the analysis of browser fingerprints. This approach allows us to represent the complex variability of browser fingerprints in a structured, hierarchical manner, moving beyond flat lists of attributes. By leveraging the feature model's, BrowserFM provides powerful analytical capabilities for a deeper comprehension of how browser features and configurations influence fingerprint components. For instance, it enables the dynamic selection of attributes crucial for fingerprint uniqueness, identifying which parts are most distinctive. This also facilitates efficient verification processes, confirming if a given fingerprint conforms to known browser configurations or if it represents an anomaly. Through our experiments, we identified that certain attributes, such as those related to WebGL rendering, exhibit high entropy. This means they vary significantly across different browser configurations, making them particularly effective for precisely distinguishing one configuration from another. Furthermore, BrowserFM demonstrated that a carefully selected, reduced set of high-discriminatory attributes can still yield meaningful insights, dramatically improving storage efficiency and making large-scale dataset analysis more manageable. This work provides a structured and efficient framework for understanding the intricate relationships between browser features, configurations, and their unique fingerprints, which is crucial for developing more effective privacy protection mechanisms, such as smart fingerprint randomization or the detection of spoofed fingerprints.

6.1.4 Open Science and Reproducibility

The research datasets, tools, and methodologies developed in this thesis are released as open-source resources to support reproducible research. We provide comprehensive reproduction instructions and technical documentation detailed in Chapter 5 to enable other researchers to replicate and extend our findings. Our commitment to open science in our research enables a better understanding of the phenomenon of browser fingerprinting by the wider community.

6.2 Research Impact and Implications

This research has different implications across multiple domains including privacy protection, browser development, security assessment, and user empowerment.

6.2.1 Privacy Protection

Our findings provide quantitative evidence of how browser configuration choices affect user privacy through fingerprinting exposure. These works enable privacy-conscious users to make informed decisions about browser configurations based on empirical data rather than speculation. Privacy advocates can leverage our datasets to evaluate the effectiveness of existing privacy tools and identify configuration parameters that should be standardized or randomized. Browser developers can use our tools and findings to assess the privacy implications of new features before deployment. Our work supports the development of privacy-preserving default browser configurations that minimize fingerprinting risks while maintaining functionality. Our feature model approach could potentially contribute to the development of more sophisticated privacy protection mechanisms by providing structured representations of configuration impacts.

6.2.2 Browser Development

Our systematic analysis provides browser developers with comprehensive understanding of how configuration parameters affect the Browser Object Model. Our research offers tools for identifying unintended side-effects of new features that may increase fingerprinting surface area. Developers can leverage our stability analysis framework to prioritize privacy-preserving implementation choices during the development process. Our formal modeling approach could facilitate automated assessment of privacy risks during the browser development lifecycle, potentially supporting the integration of privacy considerations into development workflows. Browser vendors could potentially use our methodology to evaluate the privacy characteristics of their browsers and track changes over time. Our approach supports evidence-based decision-making for feature deprecation or modification when privacy risks outweigh functional benefits.

6.2.3 Configuration-Based Content Adaptation

The ability to reliably identify browser configurations from fingerprints opens opportunities for adaptive web applications that can tailor content and interfaces based on detected configurations. Web developers can use our configuration identification techniques to provide customized user experiences, such as displaying configuration-specific instructions, warnings, or optimized interfaces. For example, a web page could display targeted privacy recommendations based on the detected browser configuration, helping users understand potential privacy implications of their current setup. Organizations can leverage this capability to provide contextual information about browser compatibility, feature availability, or security considerations specific to the user's configuration. Our approach enables proactive user education by identifying configurations that may compromise privacy and suggesting appropriate modifications. This application transforms passive fingerprinting analysis into an active tool for improving user awareness and security posture.

6.2.4 Education and Awareness

The open-source tools and datasets contribute to privacy education by providing concrete, measurable examples of fingerprinting risks. Educators can use our reproducible experiments to demonstrate privacy concepts and trade-offs in computer science and cybersecurity curricula. The work raises awareness about the privacy implications of seemingly innocuous browser configuration choices. Our research provides the technical foundation that could potentially support the development of user-facing applications for privacy risk assessment and configuration guidance.

6.3 Future Work and Perspectives

Building upon the foundations established in this thesis, several promising research directions emerge that could further advance our understanding of browser fingerprinting and configuration analysis and their impact on privacy.

6.3.1 Extension to Other Browsers and Platforms

Future work should focus on creating a collaborative platform that leverages FP-Rainbow. This collaborative platform would enable individuals to generate and contribute fingerprints from a wide range of browsers across different operating systems and hardware architectures. By centralizing these locally-generated fingerprints with FP-Rainbow, this collaborative platform would facilitate a comprehensive analysis of configuration impact at a broader scale. Such a diverse dataset would foster a deeper understanding of browser differences and aid in detecting atypical configurations or environments that might compromise user privacy. Additionally, it would be beneficial to analyze fingerprints generated by privacy-oriented browsers, such as Brave, LibreWolf, Tor, and DuckDuckGo.

6.3.2 Dynamic and Temporal Analysis

Our work has demonstrated that specific configuration parameters significantly impact browser fingerprints, opening opportunities for comprehensive analysis of browser evolution and its privacy implications. Building upon the FP-Rainbow methodology, we propose a systematic approach to extract, analyze, and document all configuration parameters found in browser source code, creating the first comprehensive temporal analysis of browser fingerprint evolution.

Phased Parameter Extraction Strategy. Given the complexity of modern browser codebases, we propose a strategic phased approach described below for parameter extraction and analysis. The first phase focuses on command-line switches (over $1,500$ in recent Chromium versions), followed by V8 JavaScript engine flags,[1] browser flags (chrome://flags interface), and finally internal settings and preferences. This progression from accessible to complex parameters ensures manageable scope while leveraging automated source code analysis tools designed for Chromium's specific coding patterns.

[1] https://v8.dev/docs/source-code

Temporal and Platform Coverage Strategy. We propose analyzing major Chromium releases from the past five years (approximately 40 releases), starting with *Linux x86_64* platforms before expanding to *Windows*, *macOS*, *Android*, and other architectures. The choice to start with a controlled set of versions and a single platform allows us to first validate our methods and gather initial results, before extending our analysis to a broader range of browser versions and platforms. Automated regression detection algorithms would identify when fingerprinting attributes appear, change, or disappear, while time-series analysis could reveal correlations between browser updates, privacy initiatives, and fingerprinting evolution.

CVE Impact and Security Correlation Analysis. We can extend analysis by establishing a link between CVEs and the configuration parameters identified within the browser's source code. This would enable us to quantify the time between the implementation of a configuration parameter in the browser's source code, the discovery of its associated CVE and its patch.

Automated Documentation and Validation Framework. Beyond Peter Beverloo's work,[2] comprehensive documentation of browser configuration parameters and their fingerprinting implications remains limited. An automated documentation system combining static code analysis with dynamic testing to catalog parameter effects on the Browser Object Model, generating descriptions, value ranges, platform compatibility, version-specific information, and fingerprinting impact scores with continuous integration updates.

Implementation and Validation. To extract the configuration parameters from the browser's source code, we would employ complementary techniques including regular expression parsing, abstract syntax tree analysis, and tools like HyperDiff [204]. Each parameter undergoes validation through automated testing using the FP-Rainbow pipeline, cross-validation with existing documentation, and statistical significance testing. This approach would create an unprecedented resource for privacy researchers and browser developers, enabling evidence-based decisions about configuration privacy implications.

6.3.3 Attributes Values Distance

Our current approach treats fingerprint attributes as discrete entities without considering the semantic or structural relationships between their values. Incorporating distance metrics between attribute values could enhance fingerprint analysis, configuration identification accuracy, and anomaly detection. The heterogeneous nature of browser fingerprint attributes requires specialized distance measurement techniques.

Attribute-Specific Distance Metrics. Different attribute types require tailored approaches: edit distances (Levenshtein) and n-gram analysis for textual attributes like user agent strings, perceptual distance metrics (SSIM) or computer vision techniques for canvas and WebGL fingerprints, normalized Euclidean or Manhattan distances for numerical attributes like screen resolution or performance measurements.

Applications and Benefits. Distance metrics enable sophisticated temporal analysis of fingerprint evolution, characterizing typical browser change patterns and

[2] https://peter.sh/experiments/chromium-command-line-switches/

detecting anomalous modifications that might indicate security compromises or privacy tool usage. Statistical models of distance distributions could identify outliers representing security threats, bot activities, or spoofing attempts. Distance-based clustering could group similar attack patterns and track fingerprint families across browser versions and update cycles.

Implementation Considerations. Key challenges include scalability when computing pairwise distances across large datasets, handling missing attributes through imputation strategies, and accommodating dynamic attribute spaces across browser versions. Validation requires comprehensive studies using ground truth datasets to evaluate distance metric effectiveness and robustness under various data quality conditions.

6.3.4 Machine Learning and Data Mining

The extensive FP-Rainbow datasets present opportunities for applying machine learning and data mining techniques to improve configuration identification and discover fingerprinting patterns.

Supervised Learning for Configuration Impact Prediction. FP-Rainbow datasets provide ideal training data where fingerprint attributes serve as features and configuration parameters as labels. Suitable algorithms include Random Forest for interpretable feature importance rankings, Support Vector Machines for non-linear relationships, and deep neural networks for discovering complex attribute interactions. These predictive approaches offer advantages over comparison-based methods: better handling of partially matching fingerprints, reduced computational complexity, and enhanced robustness to noise.

Pattern Discovery and Analysis. Unsupervised techniques like density-based spatial clustering of applications with noise (DBSCAN) could identify fingerprint patterns corresponding to specific hardware configurations while flagging suspicious outliers. Association rule mining (Apriori, FP-Growth) could reveal attribute combinations that frequently occur, informing anti-fingerprinting strategies.

Implementation Considerations. Key challenges include handling high-dimensional sparse data, managing concept drift as browsers evolve, ensuring model interpretability, and maintaining real-time performance. Federated learning approaches could enable privacy-preserving training across organizations without directly sharing sensitive data, for instance, by focusing solely on the sharing of attribute names without their corresponding values.

6.3.5 Browser Fingerprinting Injection Defenses

Building upon the extensive dataset of browser fingerprints generated by FP-Rainbow, a promising future research direction involves developing and evaluating advanced browser fingerprint injection defenses. The core idea is to leverage our comprehensive understanding of fingerprint variability to enable browsers to present a different, yet plausible, fingerprint for each session. A recent web browser named Camoufox,[3] a Firefox-based browser that already implements fingerprint injection. With the

[3] https://camoufox.com/

exhaustive set of fingerprints we can generate from FP-Rainbow, it is theoretically possible to provide a unique, realistic fingerprint for virtually every browsing session. For instance, Camoufox's documentation indicates that it uses fingerprints generated by BrowserForge. It would be crucial to empirically test if the injected fingerprint truly changes each time, or if common underlying attributes persist that could still allow identification.

Injected modifications must maintain realistic relationships (GPU-CPU combinations, screen resolution-OS pairings, language-timezone correlations) to avoid detection. Using the FP-Rainbow pipeline, we could measure injection success across Browser Object Model APIs, identifying technical limitations and persistent attributes that resist spoofing.

6.4 Final Conclusion

This thesis has demonstrated that systematic empirical analysis combined with modeling approaches can significantly advance our understanding of complex privacy phenomena such as browser fingerprinting. Through the development of FP-Rainbow, we established the first comprehensive methodology for systematically exploring browser configuration impacts, generating datasets of unprecedented scale and providing quantitative evidence that configuration parameters can be identified.Our introduction of feature models to browser fingerprinting analysis, coupled with a minimal selection of attributes, achieved remarkable storage efficiency and enabling sophisticated analysis capabilities previously unavailable in this domain.

Our results show that seemingly minor configuration choices can have major privacy implications, with the configuration parameters significantly affecting fingerprints depending on browser version, emphasizing the critical importance of privacy-by-design in browser development. Our work illustrates the transformative value of open science approaches in privacy research, with all datasets, tools, and methodologies released to enable broader community validation, replication, and extension of findings.

The successful intersection of software engineering methodologies with privacy research proves highly fruitful, demonstrating how formal modeling techniques from Software Product Line Engineering can revolutionize the analysis of complex privacy phenomena. This cross-disciplinary approach opens new research avenues and suggests that privacy research can benefit significantly from established formal methods and engineering practices.

As browsers continue to evolve and new web technologies emerge, the methodologies, tools, and formal frameworks developed in this thesis provide a robust foundation for ongoing privacy assessment and protection. The FP-Rainbow pipeline enables continuous monitoring of privacy implications as browsers evolve, while the feature model approach scales to accommodate millions of fingerprints across multiple browser versions.

Beyond academic contributions, this research has immediate practical implications for browser developers seeking to minimize fingerprinting risks, privacy advocates evaluating protection mechanisms, and users making informed configuration choices. The ultimate vision of this research is to establish a new paradigm where

privacy assessment is systematic, evidence-based, and integrated into the browser development lifecycle, ultimately empowering both users and developers to make informed decisions that enhance digital privacy without sacrificing functionality.

Bibliography

[1] M. Huyghe, W. Rudametkin, and C. Quinton, "Fp-rainbow: Fingerprint-based browser configuration identification," in *Proceedings of the ACM on Web Conference 2025*, pp. 4325–4335, 2025.

[2] M. Huyghe, C. Quinton, and W. Rudametkin, "Taming the variability of browser fingerprints," in *Proceedings of the 28th ACM International Systems and Software Product Line Conference*, pp. 66–71, 2024.

[3] M. Huyghe, C. Quinton, and W. Rudametkin, "Browserfm: A feature model-based approach to browser fingerprint analysis," in *MADWeb 2025-Workshop on Measurements, Attacks, and Defenses for the Web*, pp. 1–10, 2025.

[4] OECD, *Going Digital: Shaping Policies, Improving Lives*. OECD Publishing, 2019.

[5] E. Brynjolfsson, *The second machine age: Work, progress, and prosperity in a time of brilliant technologies*, vol. 336. WW Norton & Company, 2014.

[6] A. Acquisti, L. Brandimarte, and G. Loewenstein, "Privacy and human behavior in the age of information," *Science*, vol. 347, no. 6221, pp. 509–514, 2015.

[7] D. J. Solove, "Introduction: Privacy self-management and the consent dilemma," *Harvard law review*, vol. 126, no. 7, pp. 1880–1903, 2013.

[8] T. Berners-Lee, W. Hall, J. A. Hendler, K. O'Hara, N. Shadbolt, D. J. Weitzner, *et al.*, "A framework for web science," *Foundations and Trends® in Web Science*, vol. 1, no. 1, pp. 1–130, 2006.

[9] A. Grosskurth and M. W. Godfrey, "A reference architecture for web browsers," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 661–664, IEEE, 2005.

[10] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver, "Detecting in-flight page changes with web tripwires.," in *NSDI*, vol. 8, pp. 31–44, 2008.

[11] E. W. Felten and M. A. Schneider, "Timing attacks on web privacy," in *Proceedings of the 7th ACM conference on Computer and communications security*, pp. 25–32, 2000.

[12] K. Y. Tam and S. Y. Ho, "Understanding the impact of web personalization on user information processing and decision outcomes," *MIS quarterly*, pp. 865–890, 2006.

[13] A. Goldfarb and C. Tucker, "Online display advertising: Targeting and obtrusiveness," *Marketing Science*, vol. 30, no. 3, pp. 389–404, 2011.

[14] V. Kumar and D. Shah, "Building and sustaining profitable customer loyalty for the 21st century," *Journal of retailing*, vol. 80, no. 4, pp. 317–329, 2004.

[15] N. F. Awad and M. S. Krishnan, "The personalization privacy paradox: an empirical evaluation of information transparency and the willingness to be profiled online for personalization," *MIS quarterly*, pp. 13–28, 2006.

[16] A. Acquisti and J. Grossklags, "Privacy and rationality in individual decision making," *IEEE security & privacy*, vol. 3, no. 1, pp. 26–33, 2005.

[17] S. Barth and M. D. De Jong, "The privacy paradox–investigating discrepancies between expressed privacy concerns and actual online behavior–a systematic literature review," *Telematics and informatics*, vol. 34, no. 7, pp. 1038–1058, 2017.

[18] C. Cadwalladr and E. Graham-Harrison, "Revealed: 50 million facebook profiles harvested for cambridge analytica in major data breach," *The guardian*, vol. 17, no. 1, p. 22, 2018.

[19] C. J. Hoofnagle, B. Van Der Sloot, and F. Z. Borgesius, "The european union general data protection regulation: what it is and what it means," *Information & Communications Technology Law*, vol. 28, no. 1, pp. 65–98, 2019.

[20] D. Kristol and L. Montulli, "HTTP State Management Mechanism," RFC 2109, Internet Engineering Task Force, February 1997. Request for Comments.

[21] F. Roesner, T. Kohno, and D. Wetherall, "Detecting and defending against {Third-Party} tracking on the web," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pp. 155–168, 2012.

[22] A. Barth, "HTTP State Management Mechanism," RFC 6265, Internet Engineering Task Force, April 2011. Request for Comments, Obsoletes RFC 2965.

[23] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pp. 1388–1401, Association for Computing Machinery, 2016.

[24] T. Libert, "Exposing the hidden web: An analysis of third-party http requests on 1 million websites," *International Journal of Communication October*, 2015.

[25] B. Ur, P. G. Leon, L. F. Cranor, R. Shay, and Y. Wang, "Smart, useful, scary, creepy: perceptions of online behavioral advertising," in *proceedings of the eighth symposium on usable privacy and security*, pp. 1–15, 2012.

[26] P. Leon, B. Ur, R. Shay, Y. Wang, R. Balebako, and L. Cranor, "Why johnny can't opt out: a usability evaluation of tools to limit online behavioral advertising," in *Proceedings of the SIGCHI conference on human factors in computing systems*, pp. 589–598, 2012.

[27] M. Degeling, C. Utz, C. Lentzsch, H. Hosseini, F. Schaub, and T. Holz, "We value your privacy... now take some cookies: Measuring the gdpr's impact on web privacy," 2019.

[28] D. Machuletz and R. Böhme, "Multiple purposes, multiple problems: A user study of consent dialogs after gdpr," *Proceedings on Privacy Enhancing Technologies*, vol. 2020, no. 2, pp. 481–498, 2020.

[29] P. Eckersley, "How unique is your web browser?," in *Privacy Enhancing Technologies: 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings 10*, pp. 1–18, Springer, 2010.

[30] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, "The web never forgets: Persistent tracking mechanisms in the wild," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 674–689, ACM, 2014.

[31] P. Gill, M. Arlitt, Z. Li, and A. Mahanti, "Youtube traffic characterization: a view from the edge," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pp. 15–28, 2007.

[32] G. Maier, F. Schneider, and A. Feldmann, "Nat usage in residential broadband networks," in *International Conference on Passive and Active Network Measurement*, pp. 32–41, Springer, 2011.

[33] P. Laperdrix, W. Rudametkin, and B. Baudry, "Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints," in *2016 IEEE Symposium on Security and Privacy (SP)*, (San Jose, United States), pp. 878–894, IEEE, May 2016.

[34] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *2013 IEEE Symposium on Security and Privacy*, pp. 541–555, IEEE, 2013.

[35] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine, "Browser fingerprinting: A survey," *ACM Transactions on the Web (TWEB)*, vol. 14, no. 2, pp. 1–33, 2020.

[36] T.-F. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi, "Host fingerprinting and tracking on the web: Privacy and security implications.," in *NDSS*, vol. 62, p. 66, 2012.

[37] M. Schwarz, F. Lackner, and D. Gruss, "JavaScript template attacks: Automatically inferring host information for targeted exploits," in *Proceedings 2019 Network and Distributed System Security Symposium*, Internet Society, 2019.

[38] G. Merzdovnik, M. Huber, D. Buhov, N. Nikiforakis, S. Neuner, M. Schmiedecker, and E. Weippl, "Block me if you can: A large-scale study of tracker-blocking tools," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 319–333, IEEE, 2017.

[39] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq, "Adgraph: A graph-based approach to ad and tracker blocking," in *2020 IEEE Symposium on security and privacy (SP)*, pp. 763–776, IEEE, 2020.

[40] Ł. Olejnik, G. Acar, C. Castelluccia, and C. Diaz, "The leaking battery: A privacy analysis of the html5 battery status api," in *International Workshop on Data Privacy Management*, pp. 254–263, Springer, 2015.

[41] C. F. Torres, H. Jonker, and S. Mauw, "Fp-block: usable web privacy by controlling browser fingerprinting," in *Computer Security–ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II 20*, pp. 3–19, Springer, 2015.

[42] H. Habib, J. Colnago, V. Gopalakrishnan, S. Pearman, J. Thomas, A. Acquisti, N. Christin, and L. F. Cranor, "Away from prying eyes: Analyzing usage and understanding of private browsing," in *Fourteenth symposium on usable privacy and security (SOUPS 2018)*, pp. 159–175, 2018.

[43] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy, "Fp-stalker: Tracking browser fingerprint evolutions," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 728–741, IEEE, 2018.

[44] G. Pugliese, C. Riess, F. Gassmann, and Z. Benenson, "Long-term observation on browser fingerprinting: Users' trackability and perspective," *Proceedings on Privacy Enhancing Technologies*, 2020.

[45] A. Gómez-Boix, P. Laperdrix, and B. Baudry, "Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale," in *Proceedings of the 2018 world wide web conference*, pp. 309–318, 2018.

[46] U. Iqbal, S. Englehardt, and Z. Shafiq, "Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors," 2020.

[47] Y. Cao, S. Li, and E. Wijmans, "(cross-)browser fingerprinting via OS and hardware level features," in *Proceedings 2017 Network and Distributed System Security Symposium*, Internet Society, 2017.

[48] K. S. Ball, "Surveillance society: monitoring everyday life," *Information Technology & People*, vol. 14, no. 4, pp. 406–419, 2001.

[49] M. Foucault, *Discipline and punish: The birth of the prison*. Vintage, 2012.

[50] R. V. Ericson, "Surveillance, power and modernity: Bureaucracy and discipline from 1700 to the present day.," 1991.

[51] I. Iordanou, *Venice's Secret Service: Organizing Intelligence in the Renaissance*. Oxford University Press, 2019.

[52] J. C. Torpey, *The invention of the passport: Surveillance, citizenship and the state*. Cambridge University Press, 2018.

[53] J. Gieseke, *The history of the Stasi: East Germany's secret police, 1945-1990*. Berghahn Books, 2014.

[54] M. Rajsfus, *La Police de Vichy. Les forces de l'ordre françaises au service de la Gestapo (1940-1944)*. Éditions du Détour, 2022.

[55] M. M. Aid, *The secret sentry: The untold history of the National Security Agency*. Bloomsbury Publishing USA, 2009.

[56] J. Beniger, *The control revolution: Technological and economic origins of the information society*. Harvard university press, 2009.

[57] G. T. Marx, *Windows into the soul: Surveillance and society in an age of high technology*. University of Chicago Press, 2019.

[58] W. G. Staples, *Everyday surveillance: Vigilance and visibility in postmodern life*. Bloomsbury Publishing PLC, 2013.

[59] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner, "Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[60] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext transfer protocol–http/1.0," tech. rep., 1996.

[61] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–http/1.1," tech. rep., 1999.

[62] D. Kristol and L. Montulli, "HTTP State Management Mechanism," RFC 2965, Internet Engineering Task Force, October 2000. Request for Comments, Obsoletes RFC 2109.

[63] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: large-scale evaluation of remote javascript inclusions," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 736–747, 2012.

[64] J. R. Mayer and J. C. Mitchell, "Third-party web tracking: Policy and technology," in *2012 IEEE symposium on security and privacy*, pp. 413–427, IEEE, 2012.

[65] A. Acquisti, L. Brandimarte, and G. Loewenstein, "Privacy and human behavior in the information age*," in *The Cambridge Handbook of Consumer Privacy* (E. Selinger, J. Polonetsky, and O. Tene, eds.), pp. 184–197, Cambridge University Press, 1 ed., 2018.

[66] D. J. Solove, *Understanding privacy*. Harvard university press, 2010.

[67] F. Schaub, R. Balebako, and L. F. Cranor, "Designing effective privacy notices and controls," *IEEE Internet Computing*, vol. 21, no. 3, pp. 70–77, 2017.

[68] T. Linden, R. Khandelwal, H. Harkous, and K. Fawaz, "The privacy policy landscape after the gdpr," *arXiv preprint arXiv:1809.08396*, 2018.

[69] V. Mishra, P. Laperdrix, A. Vastel, W. Rudametkin, R. Rouvoy, and M. Lopatka, "Don't count me out: On the relevance of ip address in the tracking ecosystem," in *Proceedings of The Web Conference 2020*, pp. 808–815, 2020.

[70] C. Castelluccia and A. Narayanan, "Behavioural tracking on the internet: A technical perspective," *Communications of the ACM*, vol. 55, no. 4, pp. 96–105, 2012.

[71] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan, "Web usage mining: Discovery and applications of usage patterns from web data," *Acm Sigkdd Explorations Newsletter*, vol. 1, no. 2, pp. 12–23, 2000.

[72] V. Losarwar and D. M. Joshi, "Data preprocessing in web usage mining," in *International Conference on Artificial Intelligence and Embedded Systems (ICAIES'2012) July*, pp. 15–16, 2012.

[73] P. Manils, C. Abdelberri, S. L. Blond, M. A. Kaafar, C. Castelluccia, A. Legout, and W. Dabbous, "Compromising tor anonymity exploiting p2p information leakage," *arXiv preprint arXiv:1004.1461*, 2010.

[74] B. Danaher, M. D. Smith, R. Telang, and S. Chen, "The effect of graduated response anti-piracy laws on music sales: evidence from an event study in france," *The Journal of Industrial Economics*, vol. 62, no. 3, pp. 541–553, 2014.

[75] F. Macrez and J. Gossa, "Surveillance et sécurisation: Ce que l'hadopi rate," *Revue Lamy Droit de l'immatériel*, no. 50, pp. 79–91, 2009.

[76] B. Danaher, M. D. Smith, R. Telang, and S. Chen, "The effect of graduated response anti-piracy laws on music sales: evidence from an event study in france," *The Journal of Industrial Economics*, vol. 62, no. 3, pp. 541–553, 2014.

[77] M. A. Arnold, E. Darmon, S. Dejean, and T. Penard, "Graduated response policy and the behavior of digital pirates: Evidence from the french three-strike (hadopi) law," *Available at SSRN 2380522*, 2014.

[78] History of Information, "Louis montulli ii invents the http cookie," 2024. Accessed: 2025.

[79] Wikipedia, "Lou montulli," December 2024. Accessed: December 2024.

[80] C. Purtill, "The inventor of the digital cookie has some regrets," *Quartz*, January 2022. Interview avec Lou Montulli.

[81] Wikipedia, "Http cookie," 2024. Accessed: December 2024.

[82] Google Privacy Sandbox, "What are cookies?," 2024. Technical documentation on cookie history.

[83] M. West, A. Barth, and D. Weiss, "Cookies: HTTP State Management Mechanism," Internet-Draft draft-ietf-httpbis-rfc6265bis, Internet Engineering Task Force, March 2024. Work in Progress, intended to update RFC 6265.

[84] A. Cahn, S. Alfeld, P. Barford, and S. Muthukrishnan, "An empirical study of web cookies," in *Proceedings of the 25th international conference on world wide web*, pp. 891–901, 2016.

[85] European Parliament and Council, "Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation)," 2016. Official Journal of the European Union L 119/1.

[86] European Parliament and Council, "Directive 2002/58/ec of the european parliament and of the council of 12 july 2002 concerning the processing of personal data and the protection of privacy in the electronic communications sector (directive on privacy and electronic communications)," 2002. Official Journal of the European Communities L 201/37.

[87] European Parliament and Council, "Directive 2009/136/ec of the european parliament and of the council of 25 november 2009 amending directive 2002/22/ec on universal service and users' rights relating to electronic communications networks and services," 2009. Official Journal of the European Union L 337/11.

[88] C. Utz, M. Degeling, S. Fahl, F. Schaub, and T. Holz, "(un) informed consent: Studying gdpr consent notices in the field," in *Proceedings of the 2019 acm sigsac conference on computer and communications security*, pp. 973–990, 2019.

[89] I. Sanchez-Rola, M. Dell'Amico, M. Musolesi, and L. Bilge, "Can i opt out yet? gdpr and the global illusion of cookie control," in *Proceedings of the 2019 World Wide Web Conference*, pp. 340–351, ACM, 2019.

[90] T. Urban, D. Tatang, M. Degeling, T. Holz, and N. Pohlmann, "A study on subject data access in online advertising after the gdpr," *Data and Applications Security and Privacy XXXIV*, pp. 61–79, 2019.

[91] M. Nouwens, I. Liccardi, M. Veale, D. Karger, and L. Kagal, "Dark patterns after the gdpr: Scraping consent pop-ups and demonstrating their influence," pp. 1–13, 2020.

[92] D. Bollinger, K. Kubicek, C. Cotrini, and D. Basin, "Automating cookie consent and gdpr violation detection," pp. 2893–2910, 2022.

[93] C. Matte, N. Bielova, and C. Santos, "Do cookie banners respect my choice?: Measuring legal compliance of banners from iab europe's transparency and consent framework," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 791–809, IEEE, 2020.

[94] DLA Piper, "Gdpr fines and data breach survey: January 2025," January 2025. Seventh annual edition.

[95] J. Schuh, "Building a more private web: A path towards making third party cookies obsolete." Blog post, January 2020. Chromium Blog.

[96] Google Privacy Sandbox Team, "A new path for privacy sandbox on the web." https://privacysandbox.com/news/privacy-sandbox-update/, July 2024. Accessed: 2025-07-10.

[97] N. Kumar, R. Singh, and A. Patel, "Combating web tracking: Analyzing web tracking technologies for user privacy," *Future Internet*, vol. 16, no. 10, p. 363, 2024.

[98] R. Pan and A. Ruiz-Martínez, "Evolution of web tracking protection in chrome," *Journal of Information Security and Applications*, vol. 79, p. 103643, 2023.

[99] G. Horsman, B. Findlay, J. Edwick, A. Asquith, K. Swannell, D. Fisher, A. Grieves, J. Guthrie, D. Stobbs, and P. McKain, "A forensic examination of web browser privacy-modes," *Forensic Science International: Reports*, vol. 1, p. 100036, 2019.

[100] Mozilla, "Mozilla explains: Cookies and supercookies," April 2021. Blog post.

[101] A. A. AlQahtani and E.-S. M. El-Alfy, "Anonymous connections based on onion routing: A review and a visualization tool," *Procedia Computer Science*, vol. 52, pp. 121–128, 2015. The 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015).

[102] A. K. Jadoon, W. Iqbal, M. F. Amjad, H. Afzal, and Y. A. Bangash, "Forensic analysis of tor browser: A case study for privacy and anonymity on the web," *Forensic Science International*, vol. 299, pp. 59–73, 2019.

[103] M. Bubukayr and M. Frikha, "Effective techniques for protecting the privacy of web users," *Applied Sciences*, vol. 13, no. 5, p. 3191, 2023.

[104] N. Bielova, "Web tracking technologies and protection mechanisms," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2607–2609, 2017.

[105] S. C. Boerman, S. Kruikemeier, and F. J. Zuiderveen Borgesius, "Exploring motivations for online privacy protection behavior: Insights from panel data," *Communication Research*, vol. 48, no. 7, pp. 953–977, 2021.

[106] W3Schools, "Javascript history," 2024. Technical reference on JavaScript history.

[107] Wikipedia, "Brendan eich," December 2024. Accessed: December 2024.

[108] D. Zhang, J. Zhang, Y. Bu, B. Chen, C. Sun, and T. Wang, "A survey of browser fingerprint research and application," *Wireless Communications and Mobile Computing*, vol. 2022, no. 1, p. 3363335, 2022.

[109] K. Boda, Á. M. Földes, G. G. Gulyás, and S. Imre, "User tracking on the web via cross-browser fingerprinting," in *Information Security Technology for Applications* (P. Laud, ed.), (Berlin, Heidelberg), pp. 31–46, Springer, Springer Berlin Heidelberg, 2012.

[110] R. Upathilake, Y. Li, and A. Matrawy, "A classification of web browser fingerprinting techniques," in *2015 7th International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–5, IEEE, 2015.

[111] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel, "Fpdetective: dusting the web for fingerprinters," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 1129–1140, 2013.

[112] D. Fifield and S. Egelman, "Fingerprinting web users through font metrics," in *Financial Cryptography and Data Security* (R. Böhme and T. Okamoto, eds.), vol. 8975, pp. 107–124, Springer Berlin Heidelberg, 2015. Series Title: Lecture Notes in Computer Science.

[113] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy, "{Fp-Scanner}: The privacy implications of browser fingerprint inconsistencies," in *27th USENIX Security Symposium (USENIX Security 18)*, pp. 135–150, 2018.

[114] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in html5," *Proceedings of W2SP*, 2012.

[115] O. Starov and N. Nikiforakis, "XHOUND: Quantifying the fingerprintability of browser extensions," in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 941–956, 2017. ISSN: 2375-1207.

[116] A. Sjösten, S. Van Acker, and A. Sabelfeld, "Discovering browser extensions via web accessible resources," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pp. 329–336, 2017.

[117] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. Wien, "Fast and reliable browser identification with javascript engine fingerprinting," in *Web 2.0 Workshop on Security and Privacy (W2SP)*, vol. 5, p. 4, Citeseer, 2013.

[118] T. Laor, N. Mehanna, A. Durey, V. Dyadyuk, P. Laperdrix, C. Maurice, Y. Oren, R. Rouvoy, W. Rudametkin, and Y. Yarom, "DRAWN APART : A device identification technique based on remote GPU fingerprinting," in *Proceedings 2022 Network and Distributed System Security Symposium*, Internet Society, 2022.

[119] T. Saito, K. Yasuda, T. Ishikawa, R. Hosoi, K. Takahashi, Y. Chen, and M. Zalasiński, "Estimating CPU features by browser fingerprinting," in *2016 10th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, pp. 587–592, 2016.

[120] S. Wu, S. Li, Y. Cao, and N. Wang, "Rendered private: Making {GLSL} execution uniform to prevent {WebGL-based} browser fingerprinting," in *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1645–1660, 2019.

[121] L. Trampert, C. Rossow, and M. Schwarz, "Browser-based CPU fingerprinting," in *Computer Security – ESORICS 2022*, vol. 13556, pp. 87–105, Springer Nature Switzerland, 2022. Series Title: Lecture Notes in Computer Science.

[122] S. Chalise, H. D. Nguyen, and P. Vadrevu, "Your speaker or my snooper? measuring the effectiveness of web audio browser fingerprints," in *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, (New York, NY, USA), p. 349–357, Association for Computing Machinery, 2022.

[123] A. Vastel, W. Rudametkin, and R. Rouvoy, "FP-TESTER: Automated testing of browser fingerprint resilience," in *IWPE 2018 - 4th International Workshop on Privacy Engineering*, Proceedings of the 4th International Workshop on Privacy Engineering (IWPE'18), pp. 1–5, 2018.

[124] S. Zimmeck, J. S. Li, H. Kim, S. M. Bellovin, and T. Jebara, "A privacy analysis of cross-device tracking," in *26th USENIX Security Symposium (USENIX Security 17)*, pp. 1391–1408, 2017.

[125] S. Karami, P. Ilia, and J. Polakis, "Awakening the web's sleeper agents: Misusing service workers for privacy leakage," in *Network and Distributed System Security Symposium*, 2021.

[126] J.-P. Erkkilä, "Websocket security analysis," *Aalto University School of Science*, pp. 2–3, 2012.

[127] G. Nibert, S. Tixeuil, B. Polvé, N. J. B. M'boussi, and X. S. Nguyen, "Preventing webrtc ip address leaks," in *International Conference on Risks and Security of Internet and Systems*, pp. 365–381, Springer, 2024.

[128] A. A. Salomatin, A. Y. Iskhakov, and R. V. Meshcheryakov, "Comparison of the effectiveness of countermeasures against tracking user browser fingerprints," *IFAC-PapersOnLine*, vol. 55, no. 9, pp. 244–249, 2022.

[129] P. Baumann, S. Katzenbeisser, M. Stopczynski, and E. Tews, "Disguised chromium browser: Robust browser, flash and canvas fingerprinting protection," in *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society*, WPES '16, pp. 37–46, Association for Computing Machinery, 2016.

[130] M. A. Obidat, "Canvas deceiver-a new defense mechanism against canvas fingerprinting," in *Systemics, Cybernetics and Informatics*, 2021.

[131] P. Laperdrix, W. Rudametkin, and B. Baudry, "Mitigating browser fingerprint tracking: Multi-level reconfiguration and diversification," in *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp. 98–108, IEEE, 2015.

[132] E. Trickel, O. Starov, A. Kapravelos, N. Nikiforakis, and A. Doupé, "Everyone is different: Client-side diversification for defending against extension fingerprinting," in *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1679–1696, 2019.

[133] N. Nikiforakis, W. Joosen, and B. Livshits, "PriVaricator: Deceiving fingerprinters with little white lies," in *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pp. 820–830, International World Wide Web Conferences Steering Committee, 2015.

[134] A. ElBanna and N. Abdelbaki, "Browsers fingerprinting motives, methods, and countermeasures," in *2018 International Conference on Computer, Information and Telecommunication Systems (CITS)*, pp. 1–5, IEEE, 2018.

[135] R. Fouquet, P. Laperdrix, and R. Rouvoy, "Breaking bad: Quantifying the addiction of web elements to javascript," *ACM Transactions on Internet Technology*, vol. 23, no. 1, pp. 1–28, 2023.

[136] J. A. Pereira, M. Acher, H. Martin, J.-M. Jézéquel, G. Botterweck, and A. Ventresque, "Learning software configuration spaces: A systematic literature review," *Journal of Systems and Software*, vol. 182, p. 111044, 2021.

[137] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski, "Variability-aware performance prediction: A statistical learning approach," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 301–311, 2013.

[138] E. Guégain, A. Taherkordi, and C. Quinton, "Configuration optimization with limited functional impact," in *Advanced Information Systems Engineering* (M. Indulska, I. Reinhartz-Berger, C. Cetina, and O. Pastor, eds.), (Cham), pp. 53–68, Springer Nature Switzerland, 2023.

[139] A. Metzger, C. Quinton, Z. Á. Mann, L. Baresi, and K. Pohl, "Realizing self-adaptive systems via online reinforcement learning and feature-model-guided exploration," *Computing*, 2022.

[140] C. Qian, H. Koo, C. Oh, T. Kim, and W. Lee, "Slimium: Debloating the chromium browser with feature subsetting," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 461–476, ACM, 2020.

[141] A. Vastel, W. Rudametkin, R. Rouvoy, and X. Blanc, "FP-Crawlers: Studying the Resilience of Browser Fingerprinting to Block Crawlers," in *MADWeb'20 - NDSS Workshop on Measurements, Attacks, and Defenses for the Web* (O. Starov, A. Kapravelos, and N. Nikiforakis, eds.), (San Diego, United States), Feb 2020.

[142] B. Amin Azad, O. Starov, P. Laperdrix, and N. Nikiforakis, "Web runner 2049: Evaluating third-party anti-bot services," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*, pp. 135–159, Springer, 2020.

[143] N. Andriamilanto, T. Allard, and G. Le Guelvouit, "Fpselect: low-cost browser fingerprints for mitigating dictionary attacks against web authentication mechanisms," in *Proceedings of the 36th Annual Computer Security Applications Conference*, pp. 627–642, 2020.

[144] P. Clements and L. Northrop, *Software product lines: practices and patterns*. USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[145] A. Metzger and K. Pohl, "Software product line engineering and variability management: achievements and challenges," in *Future of Software Engineering Proceedings*, FOSE 2014, (New York, NY, USA), p. 70–84, Association for Computing Machinery, 2014.

[146] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," tech. rep., 1990.

[147] P. Clements, J. D. McGregor, and S. G. Cohen, *The structured intuitive model for product line economics (SIMPLE)*. Carnegie Mellon University, Software Engineering Institute, 2005.

[148] R. Lopez-Herrejon, J. Martinez, W. K. Guez Assunção, T. Ziadi, M. Acher, and S. Vergilio, *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines*. Springer Cham, Nov 2022.

[149] F. J. Van der Linden, K. Schmid, and E. Rommes, *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Business Media, 2007.

[150] K. Pohl, G. Böckle, and F. Van Der Linden, *Software product line engineering: foundations, principles, and techniques*, vol. 1. Springer, 2005.

[151] J. Bosch, *Design and use of software architectures: adopting and evolving a product-line approach*. Pearson Education, 2000.

[152] C. Krueger, "Easing the transition to software mass customization," in *International Workshop on Software Product-Family Engineering*, pp. 282–293, Springer, 2001.

[153] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inf. Syst.*, vol. 35, pp. 615–636, 2010.

[154] D. Nešić, J. Krüger, u. Stănciulescu, and T. Berger, "Principles of feature modeling," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, (New York, NY, USA), p. 62–73, Association for Computing Machinery, 2019.

[155] K. Czarnecki and A. Wąsowski, "Feature diagrams and logics: There and back again," *11th International Software Product Line Conference (SPLC 2007)*, pp. 23–34, 2007.

[156] S. She, U. Ryssel, N. Andersen, A. Wasowski, and K. Czarnecki, "Efficient synthesis of feature models," *Information and Software Technology*, vol. 56, no. 9, pp. 1122–1143, 2014. Special Sections from "Asia-Pacific Software Engineering Conference (APSEC), 2012" and " Software Product Line conference (SPLC), 2012".

[157] R. E. Lopez-Herrejon, J. A. Galindo, D. Benavides, S. Segura, and A. Egyed, "Reverse engineering feature models with evolutionary algorithms: An exploratory study," in *International Symposium on Search Based Software Engineering*, 2012.

[158] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed, "On extracting feature models from sets of valid feature combinations," in *Fundamental Approaches to Software Engineering* (V. Cortellessa and D. Varró, eds.), (Berlin, Heidelberg), pp. 53–67, Springer Berlin Heidelberg, 2013.

[159] M. Acher, B. Baudry, P. Heymans, A. Cleve, and J.-L. Hainaut, "Support for reverse engineering and maintaining feature models," in *Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '13, (New York, NY, USA), Association for Computing Machinery, 2013.

[160] N. Dintzner, A. van Deursen, and M. Pinzger, "Analysing the linux kernel feature model changes using fmdiff," *Software & Systems Modeling*, vol. 16, pp. 55–76, 2017.

[161] C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel, "Distance-based sampling of software configuration spaces," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 1084–1094, 2019.

[162] Y. Xiang, X. Yang, H. Huang, Z. Huang, and M. Li, "Sampling configurations from software product lines via probability-aware diversification and sat solving," *Automated Software Engineering*, vol. 29, no. 2, p. 54, 2022.

[163] F. Canales, G. Hecht, and A. Bergel, "Optimization of java virtual machine flags using feature model and genetic algorithm," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, ICPE '21, pp. 183–186, Association for Computing Machinery, 2021.

[164] U. Ryssel, J. Ploennigs, and K. Kabitzsch, "Extraction of feature models from formal contexts," in *Software Product Lines Conference*, 2011.

[165] M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, and P. Lahire, "On extracting feature models from product descriptions," *Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems*, pp. 45–54, 2012.

[166] N. Weston, R. Chitchyan, and A. Rashid, "A framework for constructing semantically composable feature models from natural language requirements," in *Software Product Lines Conference*, 2009.

[167] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki, "Reverse engineering feature models," *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 461–470, 2011.

[168] M. D. Ernst, G. J. Badros, and D. Notkin, "An empirical analysis of c preprocessor use," *IEEE Transactions on Software Engineering*, vol. 28, no. 12, pp. 1146–1170, 2002.

[169] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski, "Evolution of the linux kernel variability model," in *International Conference on Software Product Lines*, pp. 136–150, Springer, 2010.

[170] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem," in *Proceedings of the sixth conference on Computer systems*, pp. 47–60, 2011.

[171] E. Kuiter, C. Sundermann, T. Thüm, T. Hess, S. Krieter, and G. Saake, "How configurable is the linux kernel? analyzing two decades of feature-model history," *ACM Trans. Softw. Eng. Methodol.*, Apr. 2025. Just Accepted.

[172] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 105–114, 2010.

[173] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[174] H. Gomaa, "Designing software product lines with uml 2.0: From use cases to pattern-based software architectures," in *ICSR*, vol. 4039, p. 440, 2006.

[175] C. Szyperski, D. Gruntz, and S. Murer, *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[176] G. Kiczales, "Aspect-oriented programming," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4es, pp. 154–es, 1996.

[177] M. Mezini and K. Ostermann, "Variability management with feature-oriented programming and aspects," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6, pp. 127–136, 2004.

[178] D. Batory, "Feature models, grammars, and propositional formulas," in *International Conference on Software Product Lines*, pp. 7–20, Springer, 2005.

[179] S. Apel, C. Kastner, and C. Lengauer, "Featurehouse: Language-independent, automated software composition," in *2009 IEEE 31st International Conference on Software Engineering*, pp. 221–231, IEEE, 2009.

[180] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *Computer*, vol. 41, no. 4, pp. 93–95, 2008.

[181] C. Cetina, P. Giner, J. Fons, and V. Pelechano, "Autonomic computing through reuse of variability models at runtime: The case of smart homes," *Computer*, vol. 42, no. 10, pp. 37–43, 2009.

[182] J. Swanson, M. B. Cohen, M. B. Dwyer, B. J. Garvin, and J. Firestone, "Beyond the rainbow: self-adaptive failure avoidance in configurable systems," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), p. 377–388, Association for Computing Machinery, 2014.

[183] M. Cashman, J. Firestone, M. B. Cohen, T. Thianniwet, and W. Niu, "Dna as features: Organic software product lines," in *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*, SPLC '19, (New York, NY, USA), p. 108–118, Association for Computing Machinery, 2019.

[184] M. Perry, E. Clark, S. J. Murdoch, and G. Koppen, "The design and implementation of the tor browser [draft]," technical report, The Tor Project, 2013.

[185] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting malicious behavior in browser extensions," in *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 641–654, 2014.

[186] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 561–570, 2011.

[187] S. R. Choudhary, M. R. Prasad, and A. Orso, "Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 171–180, IEEE, 2012.

[188] M. Butkiewicz, H. V. Madhyastha, and V. Sekar, "Characterizing web page complexity and its impact," *IEEE/Acm Transactions On Networking*, vol. 22, no. 3, pp. 943–956, 2013.

[189] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 284–294, 2015.

[190] S. Apel, D. Batory, C. Kästner, and G. Saake, "Feature-oriented software product lines," 2013.

[191] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulesza, and P. Borba, "Investigating the safe evolution of software product lines," in *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, pp. 33–42, 2011.

[192] C. Seidl, I. Schaefer, and U. Aßmann, "Integrated management of variability in space and time in software families," in *Proceedings of the 18th International Software Product Line Conference-Volume 1*, pp. 22–31, 2014.

[193] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski, "A survey of variability modeling in industrial practice," in *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems*, pp. 1–8, 2013.

[194] K. Schmid, "Variability modeling for distributed development–a comparison with established practice," in *International Conference on Software Product Lines*, pp. 151–165, Springer, 2010.

[195] M. A. Laguna and Y. Crespo, "A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring," *Science of Computer Programming*, vol. 78, no. 8, pp. 1010–1034, 2013.

[196] P. Laperdrix, G. Avoine, B. Baudry, and N. Nikiforakis, "Morellian analysis for browsers: Making web authentication stronger with canvas fingerprinting," in *Detection of Intrusions and Malware, and Vulnerability Assessment* (R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, eds.), Lecture Notes in Computer Science, pp. 43–66, Springer International Publishing, 2019.

[197] K. Solomos, P. Ilia, S. Karami, N. Nikiforakis, and J. Polakis, "The dangers of human touch: fingerprinting browser extensions through user actions," in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 717–733, 2022.

[198] A. Shusterman, Z. Avraham, E. Croitoru, Y. Haskal, L. Kang, D. Levi, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Website fingerprinting through the cache occupancy channel and its real world practicality," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2042–2060, 2021.

[199] A. Metzger, C. Quinton, Z. Á. Mann, L. Baresi, and K. Pohl, "Feature model-guided online reinforcement learning for self-adaptive services," in *Service-Oriented Computing* (E. Kafeza, B. Benatallah, F. Martinelli, H. Hacid, A. Bouguettaya, and H. Motahari, eds.), (Cham), pp. 269–286, Springer International Publishing, 2020.

[200] U. Iqbal, S. Englehardt, and Z. Shafiq, "Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors," *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 1143–1161, 2020.

[201] M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire, and P. Merle, "Feature model differences," in *Advanced Information Systems Engineering* (J. Ralyté, X. Franch, S. Brinkkemper, and S. Wrycza, eds.), (Berlin, Heidelberg), pp. 629–645, Springer Berlin Heidelberg, 2012.

[202] C. Kröher, L. Gerling, and K. Schmid, "Comparing the intensity of variability changes in software product line evolution," *Journal of Systems and Software*, vol. 203, p. 111737, 2023.

[203] A. Gómez-Boix, D. Frey, Y.-D. Bromberg, and B. Baudry, "A collaborative strategy for mitigating tracking through browser fingerprinting," in *Proceedings of the 6th ACM Workshop on Moving Target Defense*, pp. 67–78, 2019.

[204] Q. Le Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jézéquel, "Hyperdiff: Computing source code diffs at scale," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 288–299, 2023.