

Browser Fingerprinting : Exploring Device Diversity to Augment Authentification and Build Client-Side Countermeasures

Pierre Laperdrix

▶ **To cite this version:**

Pierre Laperdrix. Browser Fingerprinting : Exploring Device Diversity to Augment Authentification and Build Client-Side Countermeasures. Cryptography and Security [cs.CR]. INSA de Rennes, 2017. English. NNT : 2017ISAR0016 . tel-01729126

HAL Id: tel-01729126
https://theses.hal.science/tel-01729126

Submitted on 12 Mar 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Thèse



UNIVERSITE BRETAGNE LOIRE

THESE INSA Rennes
sous le sceau de l'Université Bretagne Loire
pour obtenir le titre de
DOCTEUR DE L'INSA RENNES
Spécialité : Informatique

présentée par
Pierre LAPERDRIX
ECOLE DOCTORALE : MathSTIC
LABORATOIRE : IRISA

Browser Fingerprinting: Exploring Device Diversity to Augment Authentication and Build Client-Side Countermeasures

Thèse soutenue le 03.10.2017
devant le jury composé de :

Ludovic ME
Professeur, CentraleSupélec / *président*
Frank PIESSENS
Professeur, KU Leuven / *rapporteur*
Claude CASTELLUCCIA
Directeur de recherche, INRIA Grenoble / *rapporteur*
Nataliia BIELOVA
Chargé de recherche, INRIA Sophia Antipolis / *examinateur*
Gildas AVOINE
Professeur, INSA Rennes / Co-encadrant de thèse
Benoit BAUDRY
Directeur de recherche, INRIA Rennes / Directeur de thèse

Browser Fingerprinting: Exploring Device Diversity to Augment Authentication and Build Client-Side Countermeasures

Pierre LAPERDRIX

Document protégé par les droits d'auteur

Résumé en français

Contexte

L'arrivée de l'Internet a bousculé notre société à l'aube du 21e siècle. Nos habitudes se sont métamorphosées pour prendre en compte cette nouvelle manière de communiquer et de partager avec le monde. Grâce aux technologies qui en constituent ses fondations, le web est une plateforme universelle. Que vous utilisiez un PC de bureau sous Windows, un PC portable sous MacOS, un serveur sous Linux ou une tablette sous Android, chacun a les moyens de se connecter à ce réseau de réseaux pour partager avec le monde. Pour optimiser au maximum l'expérience de navigation de chaque utilisateur, les navigateurs web partagent des informations sur leur environnement. La résolution d'écran, le système d'exploitation ou le fuseau horaire sont quelques exemples d'attributs qui sont accessibles librement au sein d'un navigateur web. En 2010, grâce au site Panopticlick, Peter Eckersley de l'Electronic Frontier Foundation (EFF) a enquêté sur la diversité des informations accessibles à partir d'un navigateur web. En collectant une liste d'une dizaine d'attributs, il a été capable d'identifier de façon unique 83.6% des navigateurs dans un échantillon de 470 161. Ce pourcentage est plus élevé si les plugins Flash et Java sont installés, car ils permettent d'accéder à encore plus d'informations. Le domaine du *browser fingerprinting* est né de cette étude. A partir des données qu'il a récupérées, Eckersley a montré qu'il y avait une très grande diversité de terminaux connectés à l'Internet. Il en a conclu que cette technique pourrait être utilisée pour identifier de façon unique un usager sur Internet, contournant ainsi tous les systèmes de protection actuels contre le traçage sur Internet. Comme le *browser fingerprinting* est une technique complètement invisible et transparente, l'utilisateur n'a aucun moyen de s'y opposer. Il n'est pas non plus au courant quand un tel processus s'exécute dans son navigateur. L'impact d'une telle pratique sur le respect de la vie privée en ligne est alors très important.

Sept ans se sont écoulés depuis l'étude initiale d'Eckersley. Le *browser fingerprinting* a dépassé le stade d'un simple prototype de laboratoire et est devenu un domaine à part entière. La recherche sur ce sujet est aujourd'hui alimentée par le monde académique mais aussi par des acteurs industriels. Les piliers du web l'utilisent pour renforcer la sécurité de leurs services. Les agences publicitaires y voient une alternative pour traquer les usagers sur Internet. Les législateurs et les organisations non gouvernementales le considèrent déjà comme une menace pour la vie privée. Les personnes responsables de la sûreté de systèmes et de réseaux critiques sont conscients des possibilités d'attaques ciblées émanant de cette collecte d'informations. L'impact économique et sociétal du *browser fingerprinting* s'est très largement étendu ces dernières années mais de nombreuses questions sur le sujet se trouvent encore sans réponses. Ce qui est possible avec le *browser fingerprinting* aujourd'hui est très différent de ce qui était possible en 2010. Les organisations de standardisation du web et les développeurs de navigateurs repoussent constamment les limites du web et, par la même occasion, façonnent les mécanismes derrière cette technique.

Le *browser fingerprinting* est un domaine fascinant qui en est encore à ses balbutiements. Avec cette thèse, nous contribuons à l'écriture des premières pages de son histoire en fournissant une vue d'ensemble du domaine, de ses fondations jusqu'à l'impact des nouvelles technologies du web sur cette technique. Nous nous tournons aussi vers le futur en explorant une nouvelle facette du domaine pour améliorer la sécurité des comptes sur Internet dans une période où le nombre d'attaques en ligne sur les bases de données ne cesse de croître.

Comprendre le *browser fingerprinting* avec AmIUnique

Depuis l'étude d'Eckersley qui a révélé le *fingerprinting* en 2010, les navigateurs web ont subi d'importantes transformations. Le développement de HTML5 a repoussé les limites de ce qui est possible sur Internet avec l'introduction de nouvelles interfaces de programmation comme le Canvas, WebGL ou Web Audio. La navigation sur mobiles a aussi explosé dans la dernière décennie avec la démocratisation des smartphones et des tablettes. Pour faire suite à l'étude Panopticlick de 2010 et pour comprendre le statut actuel du *browser fingerprinting* avec toutes ces évolutions récentes, nous avons lancé en Novembre 2014 le site AmIUnique.org. Quand un utilisateur s'y connecte, le serveur récupère l'empreinte du navigateur et la compare à toutes les empreintes récoltées précédemment. L'objectif du site web est double :

- Récupérer de vraies empreintes pour comprendre leur composition et pour détecter les attributs les plus discriminants
- Informer les utilisateurs du *fingerprinting* et plus généralement du traçage sur Internet

Aux termes de cette étude en mai 2017, plus de 385 000 empreintes ont été récoltées. En février 2016, nous avons réalisé une analyse de 118 934 empreintes pour comprendre la diversité des appareils connectés à Internet, et certains des résultats obtenus nous ont surpris.

Comparaison avec Panopticlick

Pour comparer le jeu de données obtenues par le site AmIUnique avec celui de Panopticlick, nous avons calculé l'entropie de chaque attribut collecté. Les résultats ont été immédiats : on observe un niveau d'entropie similaire entre les deux études sauf pour la liste des polices de caractère et la liste des plugins. Ces chiffres s'expliquent par l'absence de plugins et de Flash sur les terminaux mobiles.

Impact des nouveaux attributs

Grâce aux données récupérées sur AmIUnique, nous avons été les premiers à enquêter sur l'impact de l'interface Canvas et WebGL. Pour les tests s'appuyant sur des rendus de canvas, l'entropie est dans le top 5 des attributs les plus discriminants. La force du *canvas fingerprinting* vient du fait qu'il s'appuie sur différentes couches du système : le navigateur, le système d'exploitation et même le matériel de l'appareil étudié. Pour les tests s'appuyant sur l'interface WebGL, on peut récupérer des informations exactes sur le modèle de la carte graphique utilisé. En revanche, le défaut majeur de cet attribut est qu'il n'est pas disponible dans tous les navigateurs. Bien que Chrome y donne accès par défaut, le navigateur Firefox masque la vraie valeur pour des raisons de sécurité.

Fingerprinting de terminaux mobiles

La question de la viabilité du browser fingerprinting sur des terminaux mobiles a été également étudiée. Vu que les utilisateurs ne conçoivent pas eux-mêmes leurs smartphones, existe-t-il une diversité suffisante des appareils mobiles pour pouvoir différencier un utilisateur d'un autre ? Si deux personnes achètent exactement le même modèle de téléphone, est-il possible de les distinguer ? À notre grande surprise, la réponse est oui. Cependant, là où les plugins et les polices de caractères étaient très discriminants sur les PCs de bureau, c'est une tout autre histoire pour les terminaux mobiles. Tout d'abord, les en-têtes user-agent sont beaucoup plus précis et détaillés sur smartphones. En plus de renseigner la plateforme et la version exacte du navigateur, certains appareils indiquent la version exacte du firmware utilisé. Certaines applications ajoutent même de nouvelles informations auxquelles un simple navigateur n'a normalement pas accès. Un exemple vient de l'application Facebook qui ajoute l'opérateur de téléphonie de l'utilisateur dans le user-agent. Toutes ces informations contenues dans le user-agent sont d'autant plus de données qui peuvent servir à différencier un appareil d'un autre. Enfin, le *canvas fingerprinting* sur terminaux mobiles est plus discriminant que sur PCs grâce à la présence d'émojis. Il n'est pas rare de voir un constructeur de téléphones fournir sa propre police d'émojis. Cette petite image qui représente une émotion facilite encore plus la distinction entre deux appareils.

Concevoir des contremesures adaptées

Défis

Une contremesure est une solution qui empêche l'identification de terminaux par la collecte d'empreintes. Concevoir une contremesure contre le *browser fingerprinting* est difficile, car il faut trouver le juste équilibre entre protection et utilisabilité. Nous listons ci-dessous les propriétés à respecter pour qu'une contremesure soit efficace :

- La contremesure ne doit pas empêcher la navigation. Comme une empreinte de navigateur reflète l'environnement logiciel et matériel d'un appareil, la modifier arbitrairement peut tout simplement empêcher son utilisateur de naviguer sur Internet. Les développeurs doivent donc restreindre le champ d'action de leur solution pour éviter de tels problèmes.

- La contremesure ne doit pas être détectable. Pour être efficace, aucun site tiers ne doit détecter qu'une solution contre le *fingerprinting* est active. Notamment, un effort tout particulier doit être fait sur la couverture des éléments modifiés. Si la couverture n'est pas totale, la contremesure est alors détectable. Par exemple, la plateforme d'un appareil est présente à la fois dans l'en-tête user-agent mais aussi dans la propriété JavaScript *platform*. Si une solution modifie l'un mais pas l'autre, cela va créer une incohérence détectable dans l'empreinte collectée et l'utilisateur sera alors plus visible que s'il n'utilisait aucune protection.

- La contremesure doit empêcher l'identification d'un appareil sur plusieurs jours. Pour être efficace, l'empreinte présentée doit changer suffisamment souvent pour empêcher de suivre un utilisateur sur Internet.

- La contremesure doit fonctionner de façon transparente. Pour fournir la meilleure expérience utilisateur possible, il ne doit pas être nécessaire de programmer des procédures de protection complexes. La solution de protection doit agir de la façon la plus automatique possible pour qu'un utilisateur sans connaissances particulières puisse protéger ses activités sur Internet.

Suivre ces procédures pendant la conception d'une contremesure est compliqué. Les solutions présentes dans l'état de l'art le prouvent, car certaines fournissent une très bonne protection mais un confort de navigation très limité. D'autres ont un champ d'action très restreint mais produisent des empreintes incohérentes et donc détectables par un site tiers. Avec Blink et FPRandom, nous apportons des solutions nouvelles pour répondre à ces problèmes.

Blink

Comme décrit dans la section précédente, de nombreuses protections contre le *browser fingerprinting* présentent un problème de couverture d'attributs. Les navigateurs évoluent aujourd'hui à une telle vitesse qu'il est très compliqué de maintenir à jour des solutions de protection. Avec Blink, nous adoptons une approche radicalement différente de celles déjà existantes. Au lieu de modifier artificiellement le contenu d'une empreinte de navigateur, nous synthétisons de toutes pièces des environnements de navigation aléatoires et uniques. L'avantage de cette approche est qu'elle ne crée aucune incohérence dans l'empreinte, car elle s'appuie sur de vrais composants qui tournent réellement sur l'appareil de l'utilisateur. Grâce à une base de données composée de 4 systèmes d'exploitation, 6 navigateurs, 39 plugins et 2762 polices de caractères, Blink synthétise des environnements de navigation très diversifiés en assemblant des composants à la volée. Blink empêche donc le suivi d'un utilisateur entre deux sessions de navigation, car les empreintes présentées seront tellement différentes d'une session à l'autre qu'aucun site tiers ne pourra les lier. En revanche, le défaut majeur de Blink est qu'il occupe un espace disque non négligeable. Pour rendre cette approche possible, il faut que la base de données soit physiquement présente sur le disque de l'utilisateur et comparée au poids d'un simple navigateur, la solution présentée est malheureusement beaucoup plus lourde.

FPRandom

Une des propriétés clés du *browser fingerprinting* concerne la stabilité des attributs collectés. Plusieurs heures, plusieurs jours, voire même plusieurs semaines peuvent s'écouler avant d'observer un changement dans l'empreinte d'un navigateur. Cette faible fréquence de changement permet l'identification, car un script de *fingerprinting* s'attend à ce que deux empreintes identiques collectées sur deux sites différents proviennent du même appareil.

Avec FPRandom, nous introduisons de l'aléa dans certaines fonctions du navigateur pour rendre certains attributs instables et inutilisables pour de l'identification. Notre prototype sous Firefox s'attaque aux 3 techniques suivantes : *le canvas fingerprinting*, *l'AudioContext fingerprinting* et l'identification du navigateur par l'ordre des propriétés d'un objet JavaScript. Notre approche est possible pour les raisons suivantes :

- Les implémentations de JavaScript dans les navigateurs sont déterministes. Par effets de bord, ce déterminisme introduit des comportements détectables et utilisables pour du *fingerprinting*. En analysant la spécification ECMAScript, nous avons identifié des endroits où ce déterminisme peut être relâché, réduisant ainsi les effets de bord du navigateur.

- Les rendus multimédias dans les navigateurs peuvent être légèrement modifiés à chaque exécution sans dégrader l'expérience utilisateur. Par exemple, la couleur de quelques pixels d'un rendu de canvas peut être changée pour introduire de l'instabilité tout en ayant un impact minimal sur l'utilisateur. Le défi ici est de trouver des modifications qu'aucun internaute ne remarquera mais qui produiront des valeurs différentes à chaque exécution d'un script de *fingerprinting*.

FPRandom exploite la flexibilité des navigateurs modernes pour rendre des attributs instables et inutilisables.

Utiliser le fingerprinting pour de l'authentification

Défis

Sur le papier, le *browser fingerprinting* devrait être un candidat idéal pour renforcer l'authentification sur Internet. Quand un utilisateur se connecte à son compte, le serveur pourrait récupérer l'empreinte de l'appareil et vérifier qu'elle est connue du système. Cependant, il y a plusieurs problèmes qui empêchent l'adoption de cette approche à grande échelle.

- Tous les attributs d'une empreinte peuvent être modifiés. Avec une simple extension de navigateur, n'importe quel utilisateur peut modifier le contenu d'une empreinte pour raconter l'histoire qu'il veut. Cela ouvre la porte à de nombreuses attaques, car un adversaire peut imiter l'empreinte d'une de ses victimes pour passer outre le système d'authentification.

- Les attributs présents dans une empreinte de navigateur sont connus des développeurs mais aussi des attaquants. Comme tous les scripts de *fingerprinting* s'appuient sur les mêmes tests, il est très facile pour un adversaire de collecter une empreinte complète.

- Les empreintes évoluent à travers le temps. Si une différence existe entre deux empreintes, est-ce qu'elle provient d'une mise à jour légitime du terminal de l'utilisateur ou d'un terminal complètement différent ? Dans le premier cas, le serveur autoriserait le bon appareil à se connecter au compte en ligne. Dans le deuxième cas, un attaquant pourrait exploiter cette faiblesse pour contourner le système de protection mis en place. Maîtriser l'évolution des empreintes de navigateur est difficile. Il est nécessaire de collecter des données pour comprendre quelles évolutions sont légitimes de celles qui ne le sont pas.

Pour pouvoir surmonter ces obstacles, nous devons nous tourner vers le côté dynamique du *browser fingerprinting*.

Pourquoi le canvas fingerprinting ?

Le *canvas fingerprinting* peut être utilisé pour un système d'authentification à questions/réponses. Contrairement aux autres techniques de *fingerprinting* qui collectent des attributs simples et prévisibles, le *canvas fingerprinting* est extrêmement dynamique. À chaque fois que l'utilisateur se connecte, le serveur demande au navigateur de peindre un élément Canvas de façon unique. Comme un rendu dépend des couches logicielles et matérielles d'un appareil, cet attribut est beaucoup plus compliqué à manipuler que d'autres attributs statiques d'une empreinte, ce qui en fait un candidat idéal pour de l'authentification. En changeant de tests canvas à chaque connexion, un attaquant ne pourra pas contourner notre système de protection, car il n'aura pas pu collecter la bonne réponse.

Les études Panopticlick et AmIUnique ont démontré qu'il n'était pas possible d'identifier de façon unique chaque appareil sur Terre. Cela veut dire qu'un système d'authentification à base de *canvas fingerprinting* ne peut pas être utilisé tout seul. Il est nécessaire qu'il soit intégré à un système d'authentification double facteurs (2FA) ou multi-facteurs (MFA).

Fonctionnement du schéma d'authentification

Le mécanisme d'authentification conçu est complètement transparent pour l'utilisateur. À la connexion, le serveur demande au navigateur de peindre deux éléments canvas avec des chaînes de caractères très précises qui ont toutes leur propre taille, rotation, ombre et couleur. Le premier rendu sert à vérifier la connexion courante et le deuxième sera utilisé pour la prochaine connexion. Comme le *canvas fingerprinting* est un processus très stable, le serveur s'attend à ce que le client renvoie une réponse qui soit identique au pixel près à celle envoyée lors de la précédente connexion. Si c'est le cas, l'appareil est autorisé à se connecter. Si des irrégularités dans le rendu sont observées, le test de canvas échoue et l'accès au compte est alors refusé.

Futur du *browser fingerprinting*

Le *browser fingerprinting* est un domaine fascinant qui est à la croisée entre le monde académique, l'industrie et les législateurs. Depuis son apparition il y a 7 ans, cette technique a mûri et a maintenant un vrai impact sur le web, car elle est déjà utilisée dans des situations réelles. Cependant, son futur est incertain. Comme le *browser fingerprinting* dépend fortement des technologies embarquées dans les navigateurs web modernes, il est difficile d'anticiper la façon dont ce domaine va évoluer. Les développeurs de navigateur et les organismes de standardisation développent le web à une vitesse tellement effrénée que c'est à chacun de deviner et d'imaginer la forme que prendra l'Internet dans 5 ans, 10 ans et même 20 ans. Quelle que soit la direction que prendra le *browser fingerprinting* dans les années à venir, il y a une certitude : ce domaine n'a pas fini de nous surprendre.

Abstract

Users are presented with an ever-increasing number of choices to connect to the Internet. From desktops, laptops, tablets and smartphones, anyone can find the perfect device that suits his or her needs while factoring mobility, size or processing power. ***Browser fingerprinting*** became a reality thanks to the software and hardware diversity that compose every single one of our modern devices. By collecting device-specific information with a simple script running in the browser, a server can fully or partially identify a device on the web and follow it wherever it goes. This technique presents strong privacy implications as it does not require the use of stateful identifiers like cookies that can be removed or managed by the user. In this thesis, we provide three contributions to the browser fingerprinting domain:

1. We perform **the analysis of 118,934 genuine fingerprints** from the *AmIUnique.org* website. We show that device identification is possible as 89.4% of collected fingerprints are unique. Notably, we observe that browser fingerprinting can be extended to mobile devices despite their constrained nature in terms of software and hardware.

2. We design **two countermeasures called Blink and FPRandom**. Since the information contained in a fingerprint is primarily used to tailor the browsing experience to the user's device, modifying it correctly presents great challenges as the wrong information can be detected and break the browsing session. By taking advantage of software diversity and randomness, we constantly change the values collected by tracking scripts so that they are unable to identify a device online.

3. We detail **a complete protocol based on canvas fingerprinting to augment authentication on the web**. At a time where passwords and logins are considered insecure with the numerous data breaches reported each year, we provide a new way to assert the identity of a device. By asking the browser to render very specific images in a canvas element, we can verify that a device is a known one, preventing account takeover through simple credentials theft.

Browser fingerprinting is still in its early days. As the web is in constant evolution and as browser vendors keep pushing the limits of what we can do online, the contours of this technique are continually changing. With this dissertation, we shine a light into its inner-workings and its challenges along with a new perspective on how it can reinforce account security.

Acknowledgements

When I first set foot at INRIA in late 2013, I never imagined the adventure I was about to embark on. Here, I want to thank everyone who helped and supported me as they made the past 4 years of my life memorable and such a fun ride to go through.

First, I want to thank all the members, past and present, of the entire DiverSE team. It has been an incredible pleasure to work every day in such a dynamic and friendly environment. Thanks to everyone for the great discussions and for explaining to me the latests trends in software engineering. I want to especially thank Paul Temple, Kevin Corre, Fabien Coulon and Dorian Leroy for the fun I had inside and outside the lab with them. I also want to give a big thanks to my previous office mates: Guillaume Becan in office F231, Erwan Bousse, Marcelino Rodriguez-Cancio and Alejandro Gomez-Boix in the now legendary office F234 where the true brainstorming takes place. It was a real pleasure to discuss and learn from you every day. Thanks to the McGyver of the DiverSE team, Olivier Barais, who provided much needed help when we first setup the AmIUnique website. Thanks to the external collaborators for the great discussions that helped shape this thesis in some way: Nick Nikiforakis, Alexandre Garel, Antoine Vastel and Clémentine Maurice. Thanks to Valérie Gouranton for suggesting me to do a research master. I would have never considered at the time to work into research and pursue a PhD. Outside of the lab, thanks to my friends Mathieu Berland, Charles Robin and Alexandre Carteron for the laughs, the success and the failures battling each other in online games and facing countless legions of online foes.

I am incredibly thankful to my whole family. I could not have made it this far without their love and overwhelming support. Thanks to Marie and Eugénie for being the best sisters a brother could ask for. You are always there to cheer me up and you always welcome me with open arms every time I make a stop in Paris. Thanks to my parents, Bruno and Valérie, for accompanying me and supporting me every step of the way for the past twenty five years. I am where I am today because of you and the incredible support you have given me through the years.

Thanks to Gildas Avoine for helping me during this thesis. You brought a fresh security perspective to the work I have done in this thesis and I had a great time collaborating with you.

Finally, this section would not be complete without thanking Walter Rudamtekin and Benoit Baudry. I basically did a PhD because of you as you showed me both the beautiful and fun side of doing research.

Thanks Walter for everything you did during my research internship and for the help you have given me at the start of my PhD. I learned a lot from you as you guided me in my first steps in the academic world. You helped me at my first conferences and I had a great time putting the final touches to some papers with you during crunching time in a deserted lab.

To Benoit, I do not think a simple thanks is enough for all the help you have given me. Every time I had a question, you always took the time to listen to me and answer me whether you were in your office or on the other side of the globe. I am also very grateful that you gave me opportunities early on in my thesis to meet with other researchers and discuss with the public about my work. Through this and by working with you, I learned a lot professionally and I have also grown a lot personally. Thank you for making the past 4 years such a blast! I am very grateful that I got to spend this adventure on browser fingerprinting with you.

Pierre Laperdrix, Rennes, July 2017

Contents

Résumé en français i

Abstract vii

Acknowledgements ix

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Contributions . . . 2
  - 1.2.1 Capturing the actual state of browser fingerprinting . . . 2
  - 1.2.2 Designing innovative countermeasures . . . 2
  - 1.2.3 Hardening online authentication . . . 3
- 1.3 List of scientific publications . . . 3
- 1.4 List of tools and prototypes . . . 3
- 1.5 Outline . . . 4

2 Background and Context 5
- 2.1 Context . . . 5
  - 2.1.1 Connecting the world . . . 5
  - 2.1.2 Identifying devices on the web . . . 8
- 2.2 Browser fingerprinting . . . 10
  - 2.2.1 Definition . . . 10
  - 2.2.2 Identifying a device with browser fingerprints . . . 11
  - 2.2.3 Investigating device diversity in the wild . . . 13
  - 2.2.4 Adoption on the web . . . 13
  - 2.2.5 Usage . . . 16
- 2.3 Diversity of collected information . . . 19
  - 2.3.1 HTTP headers . . . 20
  - 2.3.2 JavaScript . . . 21
  - 2.3.3 Plugins . . . 27
  - 2.3.4 Conclusion . . . 28
- 2.4 Defences against fingerprinting . . . 28
  - 2.4.1 Increasing device diversity . . . 29
  - 2.4.2 Presenting a homogeneous fingerprint . . . 32
  - 2.4.3 Decreasing the surface of browser APIs . . . 34
  - 2.4.4 Summary of existing defences . . . 35
- 2.5 Conclusion . . . 35

3 Investigating browser fingerprinting 39
- 3.1 Building a website for all browsers . . . 39
  - 3.1.1 Collecting fingerprints . . . 39
  - 3.1.2 Supporting legacy technologies . . . 40
- 3.2 Statistics on 118,934 fingerprints . . . 41
  - 3.2.1 Fingerprinting script . . . 41

3.2.2 Descriptive statistics . . . 43
3.2.3 Statistical validity of the dataset . . . 44
3.3 Fingerprinting with the most recent technologies . . . 45
3.3.1 Canvas fingerprinting . . . 45
3.3.2 WebGL fingerprinting . . . 47
3.3.3 Additional attributes . . . 48
3.4 Mobile fingerprint diversity . . . 49
3.4.1 Mobile and Desktop fingerprint comparison . . . 49
3.4.2 Comparison Mobile OS and browsers . . . 51
3.5 Assessing the impact of possible technical evolutions . . . 52
3.6 Investigating fingerprint evolution through time . . . 56
3.6.1 Collection of data . . . 56
3.6.2 Statistics . . . 56
3.7 Informing users and helping developers . . . 58
3.7.1 Users . . . 58
3.7.2 Fingerprint Central . . . 58
3.8 Conclusion . . . 59

4 Designing smart countermeasures **61**
4.1 Key properties to mitigate browser fingerprinting . . . 61
4.1.1 Generic properties . . . 61
4.1.2 Design choice in this thesis . . . 62
4.2 Multi-level reconfiguration and diversification with Blink . . . 63
4.2.1 Approach . . . 63
4.2.2 Implementation . . . 67
4.2.3 Research questions . . . 70
4.2.4 Experiment setup . . . 70
4.2.5 Results . . . 71
4.2.6 Threats to validity . . . 73
4.2.7 Discussion and further development . . . 73
4.3 Randomizing core browser objects with FPRandom . . . 74
4.3.1 Approach . . . 74
4.3.2 Implementation . . . 75
4.3.3 Evaluation . . . 78
4.3.4 Discussion and further perspectives . . . 81
4.4 Conclusion . . . 82

5 Augmenting authentication **83**
5.1 Using browser fingerprinting for authentication . . . 83
5.1.1 Challenges . . . 83
5.1.2 Exploiting the dynamic nature of canvas fingerprinting . . . 84
5.2 Tuning canvas fingerprinting for authentication . . . 84
5.2.1 Exploring a relevant parameter space . . . 85
5.2.2 Understanding canvas stability . . . 88
5.3 Canvas authentication mechanism . . . 89
5.3.1 Challenge-response protocol . . . 89
5.3.2 Integration in a MFA scheme . . . 90
5.3.3 Authentication function . . . 91
5.3.4 Implementation . . . 93
5.4 Security analysis . . . 94
5.4.1 Adversary model . . . 94
5.4.2 Replay attack . . . 94
5.4.3 Man-in-the-middle or relay attacks . . . 95
5.4.4 Preplay attack . . . 95
5.4.5 Guessing or building the right response . . . 96
5.4.6 Protection against configuration recovery . . . 98

5.5 Conclusion . . . 100

6 Conclusion **101**
6.1 The big picture . . . 101
6.2 Contributions . . . 102
6.2.1 The past, present and possible future of browser fingerprinting . . . 102
6.2.2 Improving current defence mechanisms . . . 102
6.2.3 Augmenting authentication . . . 103
6.3 Future work . . . 103
6.3.1 Detecting fingerprinting scripts . . . 103
6.3.2 Fingerprinting with a very high number of devices . . . 105
6.4 Perspectives and sustainability of browser fingerprinting . . . 105
6.5 Concluding thoughts . . . 106

A Code snippets from fingerprinting scripts **107**
A.1 Browser APIs . . . 107
A.2 Flash . . . 109

B Appendices to the AmIUnique study **111**
B.1 Additional data on all AmIUnique's attributes . . . 111
B.2 Our attempt at a WebGL test . . . 111
B.3 Additional Flash attributes . . . 112
B.4 Comparison to the Panopticlick study . . . 113
B.4.1 Distribution of fingerprints . . . 113
B.4.2 Distribution of browsers . . . 113
B.4.3 Anonymity set sizes . . . 113
B.5 Update to the 2016 study . . . 117

C Appendices to the Blink study **119**
C.1 Fingerprint attribute weights . . . 119
C.2 Fingerprint Dissimilarity . . . 119

D Appendices to the FPRandom study **123**
D.1 Analyzing differences in the AudioContext API . . . 123
D.2 Example of string comparison when ordering JavaScript properties . . . 124

E Appendices to the authentication chapter **125**
E.1 Phase 1 of the canvas API study . . . 126
E.2 Phase 2 of the canvas API study . . . 128
E.3 Phase 3 of the canvas API study . . . 130
E.4 Example of a complete MFA scheme . . . 132

List of Figures **133**

List of Tables **135**

Author's publications **137**

Tools and prototypes **139**

Bibliography **141**

Chapter 1

Introduction

1.1 Motivation

The Internet has rocketed our society into the 21st century with new ways to communicate and share around the world. Thanks to the technologies powering it, the web is a universal platform where the device you own does not matter. Whether you are on desktop computer running on Windows, a laptop running on MacOS, a server running on Linux or a tablet running on Android, each of us has the tools to get access to this vast network of networks and communicate with each other. In order to offer a comfortable browsing experience, browsers share details about their environment so that what is displayed on a screen is tailored to each device. In 2010, Peter Eckersley from the Electronic Frontier Foundation (EFF) decided to investigate the diversity of device-specific information that could transpire through a web browser and the results were striking [72]. By collecting several attributes like the screen resolution or the list of plugins, he was able to uniquely identify 83.6% of browsers in a sample of 470,161. This number increased to 94.2% if Flash or Java was installed. From this study, the process of collecting information from a web browser was coined as "browser fingerprinting" and a whole domain was born. With the data he gathered from visitors, he not only showed that there exists an incredible diversity of devices around the world but he highlighted that this very same diversity could be used as an identification mechanism on the web.

Seven years after Eckersley's initial report, browser fingerprinting has grown from a simple proof-of-concept to a fully fledged domain. Research on this topic is now fueled by both academia and the industry as its reach has extended well beyond the walls of the research lab. Major web actors use it to improve the security of their services [142, 143, 145, 151, 154]. Advertising and tracking companies see browser fingerprinting as an alternative to cookies [39, 52, 94, 95, 170]. Law makers and non-governmental organization consider it to be a threat to online privacy [79]. Members of the computer security community perceive it as a conduit for targeted attacks [133]. The economic and societal impact of browser fingerprinting has kept growing in the past few years and it is now at the centre of discussions of many different actors with very diverse backgrounds. However, many questions on the topic of browser fingerprinting are still left unanswered. Browser vendors and standard organizations are constantly pushing the limits of what is possible online and the contours of this technique are continually being redrawn. What was possible in 2010 is different than what can be done today as the online landscape has evolved a lot through the years. Mobile browsing is now a reality with the democratisation of smartphones [114]. The introduction of dynamic features in browsers like Canvas [28] or WebGL [30] is providing richer and more interactive experiences on the web. The development of HTML5 and the push to make web browsers more secure have rendered plugins obsolete [139, 169]. All these changes have a direct impact on the fingerprinting domain and researchers are experimenting to find out what can be done with all the information that can be collected via a web browser.

Browser fingerprinting is a fascinating domain. It is still in its infancy and a large part of its history has yet to be written. With this thesis, we want to contribute to the first pages of its story by painting a full picture of its past, present and possible future. We aim at understanding

the methods and tools that support the scientific foundations of browser fingerprinting. Our main objective is as follows: investigate the software mechanisms that leak the data contained in fingerprints. This investigation rely on the analysis of data, software architecture and code. To tackle this objective, we address the following research questions:

- How diverse are browser fingerprints on the web?
- How unique and revealing is each attribute?
- What is the impact of recent browser changes on fingerprinting? Is device identification still possible?
- How do the defence solutions present in the state of the art fare against the changing nature of browser fingerprints?
- Do they still provide appropriate protection several years after their conception? If not, what would be a sustainable approach?
- What is the impact of randomization or moving target defences to mitigate tracking?
- Can fingerprinting be used for other purposes other than device identification? If so, how and in which capacity?

Throughout this thesis, you will find the answers to all of these questions as they drive the current discussions of all the actors involved around fingerprinting. While the story of browser fingerprinting has no definitive ending in sight, we hope that the following chapters will bring a satisfying overview of this technique along with the key concepts that constitute the fundamentals of the domain.

1.2 Contributions

1.2.1 Capturing the actual state of browser fingerprinting

Since web browsers are continually changing, the entire fingerprinting domain keeps evolving and its effectiveness can vary with time. Capturing the actual state of browser fingerprinting is essential to understand what can be done with this technique and to pinpoint exactly its limits. By analysing 118,934 fingerprints collected on the AmIUnique.org website from November 2014 to February 2016, we explore the validity of browser fingerprinting in today's environment. We confirm the results of previous studies while observing new trends that stem from recent decisions of major browser vendors. 89.4% of collected fingerprints in our study are unique, proving that device identification is still possible 6 years after Eckersley's study despite the evolution of web browsers. We show that the introduction of new APIs like Canvas in HTML5 provides new highly discriminating attributes that can reinforce online identification. We also witness a shift in existing attributes. While the list of plugins was a highly revealing one several years ago, this is not the case anymore as plugins are being deprecated from modern browsers. With the recent booming of mobile browsing, we also demonstrate why fingerprinting is as effective on mobile devices than on desktops and laptops despite the restrained nature of hardware and software on these devices. The custom firmwares delivered by manufacturers coupled with the use of apps reveal unnecessary information to trackers online. Finally, we also anticipate how browser fingerprinting could evolve in the future by simulating plausible technological changes. We show that simple modifications could be made inside web browsers to improve online privacy without fundamentally changing the way the web currently works.

1.2.2 Designing innovative countermeasures

In the constant arms race between privacy advocates and tracking companies, we explore the ecosystem of defence solutions against browser fingerprinting. We especially highlight the shortcomings of current tools as most of them render their users more visible to trackers and we introduce two novel solutions.

Blink

With Blink, we explore the use of a moving target defence system to mitigate tracking through fingerprinting. By assembling at runtime an operating system, a browser, plugins and fonts, we create random but genuine fingerprints that are able to fool trackers online. We validate our approach with the creation of 5,000 random browsing environments. We show that Blink generates very dissimilar configurations and it can deceive a commercial fingerprinting script from BlueCava.

FPRandom

With FPRandom, we explore the use of randomization to render some attributes in a fingerprint unstable for tracking. We pinpointed locations in the source code of the browser where noise can be introduced so that tracking scripts will be presented with constantly changing values. We evaluate our approach with a prototype based on Firefox and we target the following attributes: Canvas fingerprinting, AudioContext fingerprinting and the unmasking of the browser through the order of special JavaScript objects. Our tests show that our modifications impact known fingerprinting scripts that use the targeted attributes. Our performance benchmarks indicate that the introduced overhead is very small, proving that the impact on the user experience is minimal.

1.2.3 Hardening online authentication

Since browser fingerprints can be stolen and simulated by attackers, they cannot act as a primary conduit against identity theft. Yet, we demonstrate that the use of dynamic attributes offers new perspectives for authentication to reinforce the security of online services. We show that canvas fingerprinting can act as a true prime candidate for stronger authentication on the web thanks to an incredible diversity and a negligible performance overhead. By tracking the evolution of a single canvas rendering on a thousand different devices, we show that the stability of a canvas test is sufficient to be used in the context of an authentication scheme as the mean number number of changes is very low. Our scheme fights off common attacks thanks to a challenge-response system that generates a new challenge each time the user logs in. With our approach, answers to previous questions cannot be replayed and the challenges are diverse enough so that an attacker cannot collect all the right responses beforehand.

1.3 List of scientific publications

Parts of this thesis are adapted from the following publications:

[1] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Mitigating browser fingerprint tracking: multi-level reconfiguration and diversification. In *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015)*, Firenze, Italy, May 2015.

[2] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the Beast: Diverting modern web browsers to build unique browser fingerprints. In *37th IEEE Symposium on Security and Privacy (S&P 2016)*, San Jose, United States, May 2016.

[3] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. FPRandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In *9th International Symposium on Engineering Secure Software and Systems (ESSoS 2017)*, Bonn, Germany, July 2017.

1.4 List of tools and prototypes

For this thesis, we developed different websites, tools and prototypes to collect data and test our approaches on real-world scenarios. The entirety of the source code for these tools is fully and freely available online. The complete list of developed artefacts is as follows:

- the AmIUnique website used to collect genuine data on the diversity of fingerprints on the web [4]
- the AmIUnique browser extensions used to collect data on the evolution of fingerprints through time [5, 6]
- the original Blink defence prototype based on VirtualBox which assembles random browsing environments at runtime [7]
- the second version of Blink running on Docker [8]
- the Blink browser extensions responsible for transferring user profiles between generated environments [9, 10]
- the Fingerprint Central website developed as part of the Google Summer of Code 2016 to help the Tor organization improve the fingerprinting defences of the Tor browser [11]
- the FPRandom defence prototype which introduces random noise into advanced fingerprinting routines [12]

1.5 Outline

The thesis is organised as follows.

Chapter 2 sets the stage by introducing the context of this work. Browser fingerprinting is then explained in its entirety. What are its foundations? What are the mechanisms behind it? What is collected? How can it be used? This chapter refers to scientific publications from the literature but also to online tools from developers and companies as browser fingerprinting has extended beyond the simple research prototype to be used in real-world scenarios. We finish this chapter by detailing current defences against browser fingerprinting while highlighting their shortcomings.

In Chapter 3, we present a much needed update on the actual state of device diversity with our AmIUnique.org website. By collecting browser fingerprints from online visitors, we are able to understand the evolution of the fingerprinting domain by assessing the impact of new attributes and witnessing new trends. We also report about the impact of AmIUnique beyond our research. The website raises awareness among visitors about online tracking and we invite them to improve the security of their day-to-day browsing activities by installing simple browser extensions. AmIUnique also provided the blueprint for *Fingerprint Central*, a project aimed at improving the online privacy of Tor browser's users with regards to browser fingerprinting.

Chapter 4 details our two contributions to mitigate tracking through browser fingerprinting. The first one called *Blink* works at the operating system level. With a system of reusable components, it synthesizes at runtime environments that exhibit random but genuine browser fingerprints to fool online trackers. This solution addresses one of the main problems in the current literature which is the generation of mismatches between attributes in collected fingerprints. The second called *FPRandom* operates at the browser level. It introduces noise into very specific browser routines to prevent the use of advanced fingerprinting techniques.

Chapter 5 reports on our efforts to improve account security by detailing an authentication scheme based on canvas fingerprinting. As information contained in a fingerprint can be modified and replayed, using browser fingerprints to improve the security of login systems presents great challenges against well-known attacks. We explore in this chapter the use of canvas fingerprinting as we can generate a unique set of challenges every time the user connects to the service.

Lastly, Chapter 6 concludes this thesis by summarizing our main contributions and by detailing future avenues of research and progress for browser fingerprinting.

Chapter 2

Background and Context

In this chapter, we perform an in-depth analysis of browser fingerprinting to understand how it works and where it stems from. As its foundations are rooted in software diversity, we focus on the software tools and technologies that support its exploitation and its mitigation. Moreover, as touched upon in Chapter 1, browser fingerprinting has an impact from a technical, business, privacy and security perspective. Hence, in order to get a complete view of the domain, we embrace the diversity of approaches and refer to academic publications as well as public reports and tools from the civil society and private companies.

This chapter is organised as follows. To introduce the context of this thesis, we first go back in Section 2.1 to the origins of the web to understand the underlying technologies that support such a massive network. Then, we explore in Section 2.2 the current state of browser fingerprinting by explaining what is a fingerprint, how it is built, and how it is used on the Internet. Section 2.3 reports on the different types of information that can be collected through a web browser. In Section 2.4, we look at the different tools that exist to mitigate and defend against unwanted tracking. Finally, we provide a conclusion in Section 2.5.

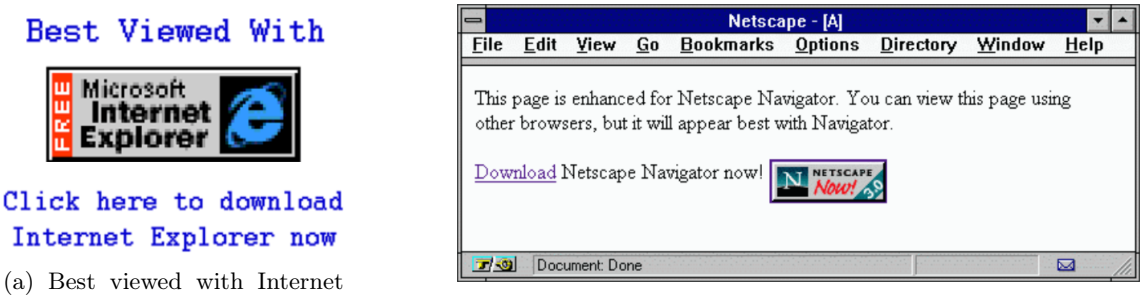
2.1 Context

2.1.1 Connecting the world

Beginning of the web and the support of device diversity

In 1989, Tim Berners-Lee outlined in a document entitled "HyperText and CERN" how he would link information systems at CERN to prevent loss of information and to accommodate the high turnover of people. He notably wrote that, to make such a system feasible, "heterogeneity" of systems must be taken into account [38]. Little did he know at the time that this early paper would fuel the research into what we know today as the World Wide Web. One of the key ideas of the early development of the web is that, for anyone to get access to this vast network of machines, it should be device agnostic, i.e. run on any device with any type of architecture. The World Wide Web Foundation details this as "Universality: For anyone to be able to publish anything on the web, all the computers involved have to speak the same languages to each other, no matter what different hardware people are using; where they live; or what cultural and political beliefs they have" [196].

HTTP and HTML were born from that need of having a universal way of communicating between machines. The Hypertext Transfer Protocol (HTTP) provided the "application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems" [165] while the Hypertext Markup Language (HTML) was "a simple markup language used to create hypertext documents that are platform independent" [164].



(a) Best viewed with Internet Explorer

(b) Best viewed with Netscape Navigator

Figure 2.1: Examples of "Best viewed with" banners

Indicating browser limitations: the case of the user-agent header

In the early 90s, web browsers started to appear from various teams around the world to support these soon-to-be standards. However, as the foundations of the web started to evolve to keep pushing what is possible online, not every browser and not every platform supported the latest additions. Some browsers conformed to only a subset of the specifications and developed their own features. This started the now infamous era of the "Best viewed with X" banners as seen in Figure 2.1.

To prevent incompatibility problems, the HTTP protocol includes the "User-Agent request-header". According to the official standard of May 1996, the header is present "for statistical purposes, the tracing of protocol violations, and automated recognition of user agents for the sake of tailoring responses to avoid particular user agent limitations" [165]. Browsers started to include their name, their version and even sometimes the platform on which they were running.

As reported by [98] and [99], the story of the user-agent header is very rich and it keeps writing itself today as modern browsers still bear the legacy of the very first browsers. The information contained in this header has become complex as browser vendors started copying the value of their competitors to declare their compatibility with a different rendering engine. For example, the current user-agent for version 57 of a Chrome browser running on Linux is the following:

```
Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome
    /57.0.2987.110 Safari/537.36
```

The only relevant pieces of information here are "(X11; Linux x86_64)" and "Chrome/57.0.2987.110". Other strings like "Gecko", "KHTML" or "Safari" are present to declare their compatibility with other layout engines. The string "Mozilla/5.0" even dates back from the time where the first ever version of Firefox was released to the public. All modern web browsers now include it in the user-agent header for no particular reason.

In the end, the user-agent header can be considered as the very first piece of information that deliberately indicated differences between devices to help developers take into account browser limitations. As we will see in the following sections, this is one of many differences that are observable from today's web ecosystem.

Bridging the gap between web browsers and native software applications

At the very beginning of the web, pages needed to be reloaded completely to allow live modifications. In 1995, Brendan Eich added a scripting language called JavaScript inside the Netscape Navigator to make web pages more dynamic. From then on, the language quickly gained a lot of traction and was implemented in most browsers in the months following its introduction. The specification of the language became standardized in June 1997 under the name "ECMAScript", with JavaScript being the most well known of its implementations at the time.

As the language started growing and as browsers started to offer more and more features to their users, developers pushed to create a bridge between the browser and the platform it is running on. The goal was to incorporate information from the user's environment inside the browser to feel more like a native software application. The very first edition of the ECMAScript specification

offers the first traces of such integration with details on the "Date" object that "contains a number indicating a particular instant in time to within a millisecond" [73]. Especially, Section 15.9.1.7 refers to "Local Time Zone Adjustment" where an "implementation of ECMAScript is expected to determine the local time zone adjustment by whatever means are available". To conform to the specification, browsers directly got from the operating system the timezone of the device. This piece of information is another example of one of the first detectable differences between devices on the web.

Now, in 2017, browsers are providing more features than even before through a wealth of Application Programming Interface (API). An API is an interface that provides an entry point to specific objects and functions. Some APIs require a permission to be accessed like the microphone or the camera but most of them are freely accessible from any JavaScript script. In the past 10 years, new APIs have surfaced to reinforce the capabilities offered by web browsers. The Canvas [28] and WebGL [30] API rely on the graphic card of the device to render 2D and 3D animations in real-time. The Web Audio API [27] provides a powerful system for controlling audio on the Web. A WebVR API [29] is being finalized to provide access to Virtual Reality devices. The modern browser has slowly shifted in the past 15 years from being a tool that displays simple HTML pages to a very rich multimedia platform compatible with many formats and devices. This transition has created a very strong link between the browser and the underlying layers of the system. As a result, the diversity of devices connected to the Internet is more exposed than before through these newly added APIs since they exploit the unique characteristics of the hardware and software that the browser rely on.

Architecture of web browsers

To conclude this section, let us take a look at the architecture of modern web browsers. This will help us get a better understanding of how everything is structured and why we can observe differences between browsers.

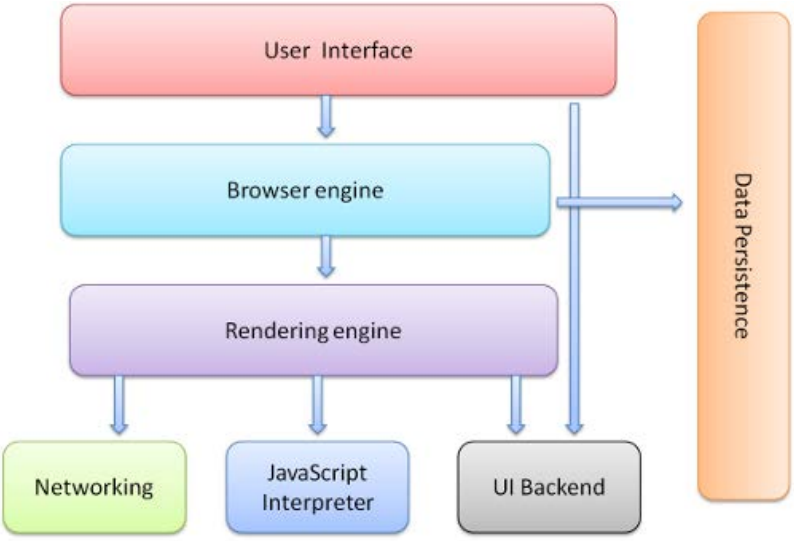


Figure 2.2: High-level structure of a web browser taken from [44]

Figure 2.2 provides a simplified view of the architecture of a web browser. The top half of the figure represents the components that handle the interaction of the user with the browser. All the elements that are in the bottom half are not directly visible to the user but they are responsible for most of the heavy lifting inside the web browser. We want to highlight specifically 4 of these components:

- The rendering engine: it is responsible for displaying web content directly in the browser. Its main role is to display and render web pages in HTML by parsing them, interpreting them and painting them. The most well-known rendering engines are Gecko for Firefox, Blink for

Chrome and WebKit for Safari. Mozilla is currently working on a more advanced rendering engine that takes advantage of modern hardware called Quantum [156].

- The JavaScript interpreter: As web pages are now populated with very large JavaScript libraries, browsers must be able to process thousands of lines of code efficiently, otherwise the user experience will deteriorate quickly. Major web actors have turned to just-in-time (JIT) compilers to combine the best of compilers and interpreters (see more details in [103]). Each browser has its own JavaScript engine: SpiderMonkey for Firefox, V8 for Chrome, Nitro for Safari or Chakra for Microsoft Edge.

- Networking: this component is responsible for making HTTP requests when contacting servers around the world. It notably adds in requests several headers that are specific to the browser as we saw with the example of the user-agent.

- UI Backend: This part uses system libraries to display and adapt a web page to the user's screen.

Based on these four core components, web browsers do not behave the same way. The HTTP requests will be different. The rendering engines will not render HTML elements identically with pixel precision. The JavaScript interpreters do not offer the same optimisations and may react differently to language quirks. The UI backend can impact how the user actually sees what is displayed on a physical screen. A difference in behaviours can also be noticed when browsers are run on devices with different hardware and software characteristics as each browser will adapt to offer the best experience possible to the user.

In the end, the way web browsers are built enables them to take full advantage of the capabilities of each device. Whether it is on the software or the hardware side, a browser is flexible enough to adapt to its environment by having separate and distinct components. However, as we will see in Section 2.3, differences between browsers and devices can be detected because of modern browsers' architecture and flexibility, thus making browser fingerprinting a reality.

2.1.2 Identifying devices on the web

Identifying a device on the web has many purposes from improving security to personalizing the online experience of each user. Here, we detail the processes that are used to perform such a detection along with the economic impact it has on the web ecosystem.

Cookies and third-party tracking

Cookies are small pieces of data sent from a website and stored directly in the browser. Any information can be stored in a cookie but the majority of them contain a unique identifier that links to a specific online identity. For example, when users log on their favourite websites, a cookie is stored to identify them and log them automatically on subsequent connections. The information in the cookie enables the website to verify the identity of each user for each request.

As a technology, cookies have been part of the Internet since the very beginning. The official specification came from RFC 2109 [162] in 1997 and RFC 2965 [163] in 2000 on the subject of "HTTP State Management Mechanism". But the very first traces date back from 1994 where a programmer named Lou Montulli came up with the idea of giving the web a memory. He coined the name "cookies" as an homage to magic cookies [200].

Nowadays, it is very hard to write about cookies without mentioning third-party tracking as they are the primary conduct for this type of practice. When a web page is loaded, many resources like scripts or images are fetched from servers that are different from the main domain that the user is visiting. For example, when a user consults an article on the BBC.com website, the browser connects to the BBC server to fetch the article but it will connect to other websites from different companies to provide additional features. One of them is Chartbeat which provides statistical measurement tools as they enable "media organizations to understand and harness genuine audience engagement" [50]. Others are more famous like Facebook or Twitter as they provide the well-known "Share" buttons on all articles [82,189]. All these servers that do not belong to the first-party domain

are called **third-party servers**. Some websites call more than a hundred of them to provide the desired services.

Figure 2.3 illustrates the concept of third-party tracking. In the example, when the browser connects to *SuperAds* to fetch the "ad.js" script, it will put a cookie with a unique ID directly in the browser. If the user connects to another website where another script is also provided by *SuperAds*, this third party has the ability to identify the user by collecting the cookie that was

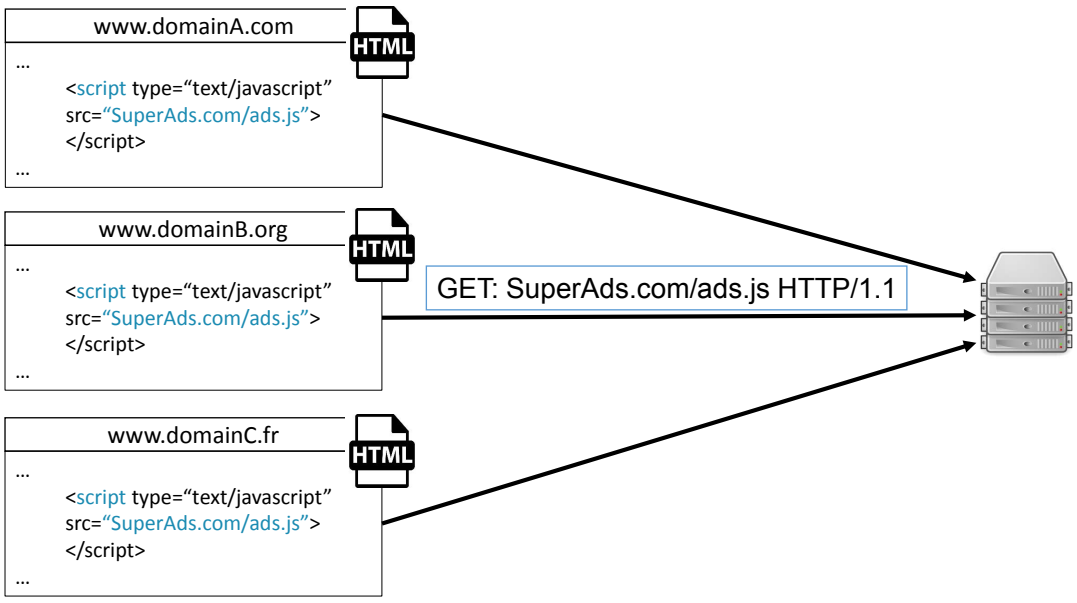


Figure 2.3: Illustration of third-party content delivery

Blocking cookies and the rise of ad blockers

One of the most important aspects of third-party tracking is that it is completely transparent as the creation of cookies happens in the background. This poses a serious privacy dilemma as most users are completely oblivious to it and they do not have any say in what happens in their browser.

In an effort to push privacy at the forefront, a European directive nicknamed the "EU Cookie law" was adopted by all EU countries in May 2011. The directive gives any individual the right to refuse the use of cookies that could reduce their online privacy. Banners asking users to allow the use of cookies started appearing everywhere on the web. However, as users became accustomed to see these banners, the impact of the law has completely evaporated. Users simply click on any buttons to make the banner or popup disappear as they want to get to the website content quickly. Some developers even released extensions like *CookiesOk* [57] or *I don't care about cookies* [58] to automatically make the banners disappear as they negatively impact the user experience. All in all, the law has become ineffective and did not bring the much wished for privacy advancements promised by lawmakers.

More and more users are now turning to alternatives to take back control of their privacy. Some refuse third-party cookies by directly modifying browser preferences as most modern browsers offer a built-in blocking mechanism. Others install blocking extensions like AdBlock [16] or Ghostery [96] to prevent tracking scripts from being executed. As reported by PageFair, a company specialised in "respectful and sustainable advertising" [135], 11% of the global Internet population is blocking ads on the web at the end of 2016 [136]. This represents 615 million devices with an observed 30% growth in a single year. The use of ad blockers is becoming more and more common as people get annoyed by intrusive online ads that can pose a threat to their privacy on the web. However, as we will see in the next section, there are hidden costs in this arms race against these invisible third-party companies.

Online advertising and the importance of tracking

Online advertising is an integral part of today's web ecosystem. As reported by the Interactive Advertising Bureau (IAB), it reached $72.5 billion in revenues in the United States alone in 2016 with a 21.8% growth from 2015 [102]. Moreover, for the first time ever, digital ad revenues surpassed TV ad revenues in the US in 2016 [188]. In the midst of this booming sector, tracking users is crucial as it enables targeted advertising. Users may be more inclined to click on an ad if it is targeted to their personal preference; thus, advertising agencies are willing to pay more money if it means that their ads are targeted to a more receptive population.

However, as we saw in the last section, more and more users are adopting ad blockers. The economic fallout from this new trend is far from being negligible as the loss of revenues from ad blockers was estimated at $21.8 billion in 2015 [137]. Many popular websites are turning to a premium model to survive as the decline in ad revenues forces them in unstable financial situations. Many newspapers have developed their own subscription-based service to survive the switch from paper to digital. New uses of social media are also impacted as some users earn a living making videos on YouTube or streaming on Twitch. This decline in revenues culminated in the past few years in an arms race between users of blocking extensions and websites. Mughees et al. reported that 1100 of the top Alexa-100k websites performed detection of ad blockers and some of them block the access to content if an ad blocker is active [120]. In August 2016, Facebook tried to circumvent popular blocking extensions but it was met with a backlash from the ad-blocking community [84, 85].

All in all, online advertising is here to stay as long as no sustainable alternatives are found. Web actors from the bigger ones like Google to the smaller ones like online newspapers all benefit from it. With the rise of ad blockers, new companies are turning to new ways to identify devices and to avoid using cookies. In this tumultuous economic and privacy context, this is where browser fingerprinting finally takes the stage.

2.2 Browser fingerprinting

2.2.1 Definition

A **browser fingerprint** is a set of information related to a user's device from the hardware to the operating system to the browser and its configuration. **Browser fingerprinting** refers to the process of collecting information through a web browser to build a fingerprint of a device. Via a simple script running inside a browser, a server can collect a wide variety of information from public interfaces called APIs and HTTP headers. Contrary to other identification techniques like cookies that rely on a unique identifier (ID) directly stored inside the browser, browser fingerprinting is qualified as completely *stateless*. It does not leave any trace as it does not require the storage of information inside the browser. If the fingerprint contains enough information, it is very likely that the fingerprint will be unique for this device, hence it can be used to fully or partially identify a visitor on a website thanks to the device characteristics that are exposed.

For the rest of this thesis, the terms "browser fingerprint" and "device fingerprint" will be used interchangeably. However, it should be noted that we focus only on information collected through a web browser. We do not cover the identification of devices through smartphone applications like Kurtz et al. [110] or Wu et al. [203] as they have access to more information than with a simple browser and they require additional permissions to get installed. We also do not focus on the analysis of the structure of network packets similar to the features offered by tools like nmap [125] as they fall out of context of what the browser has access to. Finally, even if they are touched upon in some sections of this chapter, we do not study in depth how the IP address or the geolocation of the user can contribute to the identification of a device. While they can be used to complement a fingerprint, we focus in this dissertation on what can be done entirely from the information given by a web browser as connection data can be changed via a proxy or a Virtual Private Network (VPN).

Complementarity with cookies and the lack of control mechanisms

As a legacy technology of the web, all browsers have a built-in support for cookies and they provide control mechanisms to dictate when and how cookies are used. By default, cookies are accepted on any websites but a user can change this behaviour by modifying the browser preferences or by installing an extension that will manage them in a specific way. Users have the power over these objects and they can remove them at any time.

Browser fingerprinting can be considered to be on the opposite side of the identification spectrum as it is not a part of any web standard. There is no built-in mechanism to allow or refuse the execution of fingerprinting scripts. The privacy implications are then really strong as users have no easy way of controlling this technique. They must configure their browsers precisely or install the right extension to prevent giving away too much information. Any website that a user contacts when browsing the web has the tools to build a complete fingerprint of a device and identify it. Finally, it should be noted that fingerprints do not replace cookies. They complement each other as they can be used in parallel to provide very extensive means of identification.

2.2.2 Identifying a device with browser fingerprints

Building a fingerprint

To build a browser fingerprint that is as complete as possible, one needs to rely on several browser mechanisms to collect information from different sources.

The first pieces of information that a server can collect comes from the very first interaction between the server and the client. As defined in the RFC on HTTP [166], the browser sends additional headers when it makes a request so that the server can adapt its response to the client. Figure 2.4 illustrates this very first exchange. These headers include basic information on the type of formats or encoding supported by the browser but it extends well beyond that by indicating the device platform or the language preferred by the user. At the time when RFC 2616 [166] was written, the authors were well aware of the privacy implications of revealing very specific user information, especially with the inclusion of the Accept-Language header. They wrote "Accept request-headers can reveal information about the user to all servers which are accessed. The Accept-Language header in particular can reveal information the user would consider to be of a

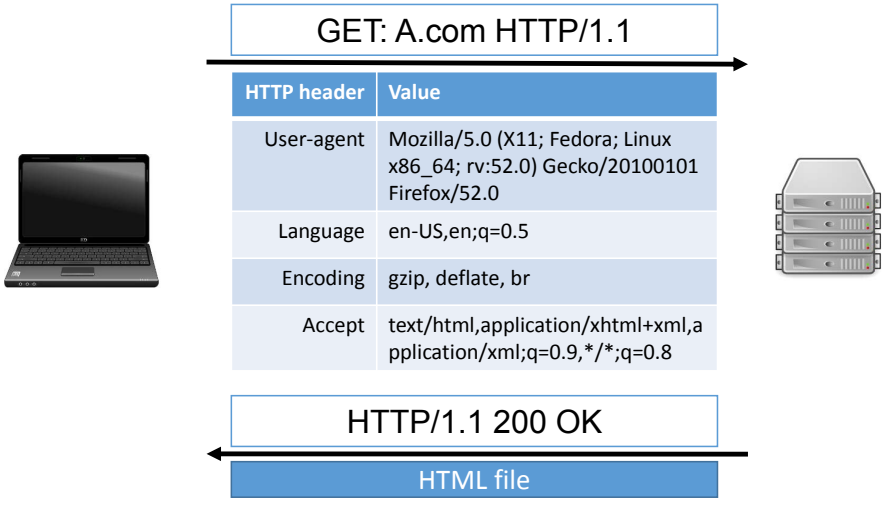


Figure 2.4: Collection of HTTP headers (passive fingerprinting)

This first stage in the collection of device-specific information is sometimes referred as **passive fingerprinting** as the HTTP headers are directly sent to the server without any interaction with the browser. This comes in contrast with **active fingerprinting** that rely on the execution of scripts inside the browser to collect the much sought after information. Figure 2.5 illustrates this second stage. In the example, the "fingerprint.js" script is not embedded in the page so the browser fetches it. When the script is received, it is directly executed and it collects information by querying different parts of the browser. All the collected information are then sent to the same remote server.

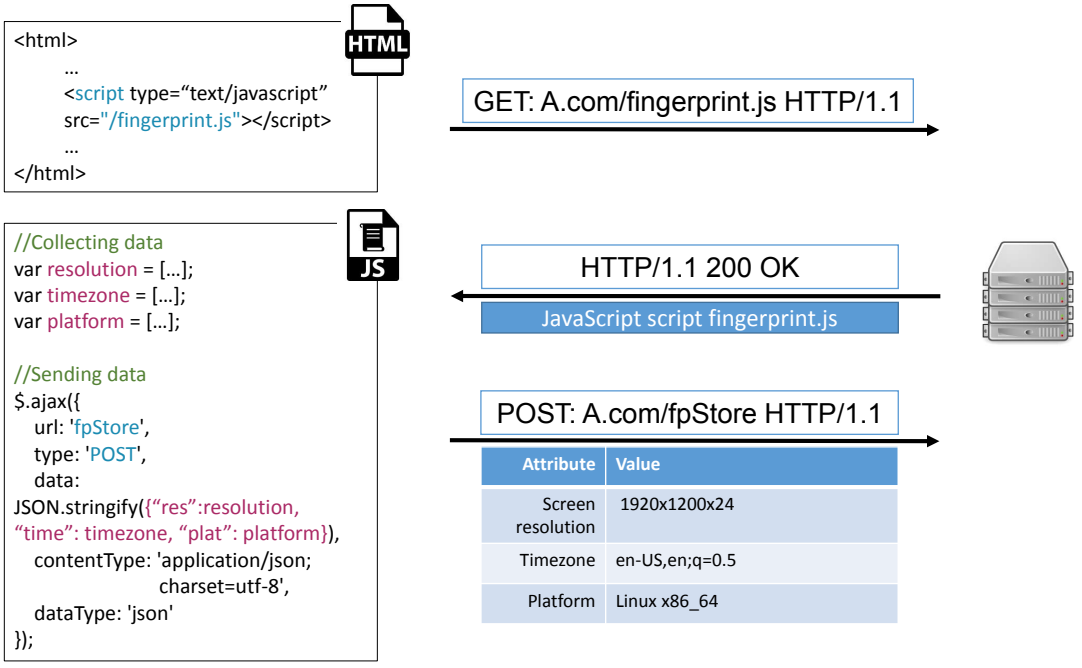


Figure 2.5: Collection of JavaScript attributes (active fingerprinting)

By default, all browsers fetch external resources present in a web page and they are executed without any warnings or permissions. This default behaviour is really important to point out as it explains why browser fingerprinting is possible today. As a fingerprinting script is loaded and executed like any other script in the browser, the collection process happens in the background without the user ever noticing. Building a browser fingerprint is then **completely transparent to the user** as the browser never asks for confirmation if a script should be executed or not.

Two key principles: uniqueness and stability

In order for a fingerprint to be used in any form of identification, the collected information must follow two key principles.

- Uniqueness: To provide enough ground for identification, each fingerprint should be as unique as possible. This does not necessarily mean that each piece of collected information must be unique. As we will see in Section 2.3, many attributes like the screen resolution or the timezone have values that can be shared by many users. But the combination of all attributes should be unique to a particular device to enable identification. Then, this uniqueness property is correlated to the size of the dataset and the distribution of the population. If a site handles several thousand connections every day, the number of collisions between fingerprints could be very low or even null compared to a site handling millions of connections. Gulyás et al. explored the concept of near-optimal fingerprinting with constraints, i.e. fingerprinting in a context where only a limited number of the user's attributes can be queried [97]. They showed that, from a dataset of 43,656 lists of fonts, they were able to uniquely identify 30.58% of users just by checking 5 fonts for each of them. This proves that most attributes may not be needed when identifying a device in a dataset.

- Stability: As users update their devices and customize their browsers, the collected fingerprints are changing. In order to *link* browser fingerprints belonging to the same device, the quantity of modified information should be as small as possible. For example, a server can expect to see an increase in the browser version from time to time but a change in the timezone is more unusual as it implies that the user is travelling. Different behaviours and different types of changes must be identified to be able to identify a device with accuracy.

In the end, while uniqueness addresses the problem of identification for a given moment in time, stability looks to solve it over time. These two key notions will be developed all along this thesis and they are also at the core of the privacy and anonymity discussion. To design strong countermeasures, a developer can work on these two dimensions to have a constantly changing signature that a server cannot follow.

2.2.3 Investigating device diversity in the wild

Related work

In 2009, Mayer investigated if device diversity could lead to the deanonymization of web clients [112]. Especially, he looked to see if differences in browsing environments could be exploited by a remote server to identify users. He noticed that a browser could present "quirkiness" that came from the operating system, the hardware and the browser configuration. He conducted an experiment where he collected the content of the *navigator*, *screen*, *navigator.plugins*, and *navigator.mimeTypes* objects of browsers who connected to the website of his experiment. Out of 1328 clients, 1278 (96.23%) could be uniquely identified with the combination of these attributes. However, he added that the small scale of his study prevented him from drawing a more general conclusion.

A year later, Peter Eckersley from the Electronic Frontier Foundation (EFF) conducted the Panopticlick experiment. By communicating on social media and popular websites, he amassed 470,161 fingerprints in the span of two weeks. Contrary to Mayer, the amount of collected fingerprints gives a much more precise picture on the state of device diversity on the web. With data from HTTP headers, JavaScript and plugins like Flash or Java, 83.6% of fingerprints were unique. If users had enabled Flash or Java, this number rose to 94.2% as these plugin provided additional device information. This study was the first to prove on a very large scale that "browser fingerprinting" was a reality. The privacy implications that emerged from it are really strong as a device with a not-so-common configuration can easily be identified on the Internet.

Our contribution with AmIUnique

Since 2010, no contributions have been made to analyse device diversity and confirm Eckersley's findings. In the span of several years, mobile browsing has been booming and new browser APIs have entered the scene but no study has been conducted to analyse and understand the impact of these changes. To address this problem, we launched in November 2014 the AmIUnique.org website to investigate device diversity in the wild. Similarly to the Panopticlick website, when users connect to AmIUnique, their device fingerprint is collected and compared with other fingerprints stored in our database. In Chapter 3, we report on the analysis that we performed on 118,934 fingerprints. We show that mobile fingerprinting is possible but for different reasons than on desktops and we also notice a change in the most discriminating attributes.

2.2.4 Adoption on the web

Research studies

Since Eckersley's study in 2010, different studies have been conducted to quantify the adoption rate of browser fingerprinting on the web.

In 2013, Nikiforakis et al. with the Cookieless Monster study [124] crawled the Alexa top 10,000 sites to look for fingerprinting scripts from the three following companies: BlueCava, Iovation, ThreatMetrix. They discovered 40 sites making use of their fingerprinting code.

The same year, Acar et al. performed a much larger crawl by visiting the top Alexa 1 million websites with the FPDetective framework [14]. While visiting a web page, they log the access to

very specific JavaScript functions and they decompile the Flash files they encounter to verify the presence of fingerprinting related function calls. With their approach, they do not need to rely on a known list of tracking scripts as they can directly look for behaviours related to fingerprinting activities. They found 404 sites out of 1 million performing JavaScript-based font probing and 95 sites out of 10,000 performing Flash-based font probing.

Finally, in 2016, two researchers at Princeton University released the OpenWPM platform, "a web privacy measurement framework which makes it easy to collect data for privacy studies on a scale of thousands to millions of site" [160]. To demonstrate the capabilities of their tool, they made an analysis of the top 1 million sites to detect and quantify emerging online tracking behaviours [77]. Their findings provide more accurate results than in the past as they instrumented extensively a very high number of JavaScript objects to build a detection criterion for each fingerprint technique. Out of 1 million, they found 14,371 sites performing canvas fingerprinting, 3,250 sites performing canvas font fingerprinting and 715 sites performing WebRTC-based fingerprinting. These numbers are much higher than what was reported in previous studies. However, they do not report on the spread of more traditional techniques like the collection of navigator properties. After contacting the authors, the amount of data they collected was so important that they needed to perform a very time-consuming process of analysing scripts by hand to design new heuristics. In the end, the number of actors performing device fingerprinting on the web is probably much higher than what is currently reported by large crawls.

Evolution of privacy policies

As browsers started to reveal larger parts of their configuration, many websites updated their privacy policy to indicate that they started collecting and storing device-specific information. Here, we take a look at the privacy policies of major web actors to see if they perform device fingerprinting and to find out what they do with this information.

- **Google** At the time of writing, the latest version of Google's privacy policy dates from March, 1st 2017 [151]. It notably includes the following pieces of information: "We collect device-specific information (such as your hardware model, operating system version, unique device identifiers, and mobile network information including phone number)" and "device event information such as crashes, system activity, hardware settings, browser type, browser language, the date and time of your request and referral URL". There is also a specific paragraph named "Cookies and similar technologies" where it is mentioned "We and our partners use various technologies to collect and store information when you visit a Google service, and this may include using cookies or similar technologies to identify your browser or device". While there is not a single mention of the term "fingerprinting", collecting device-specific information to identify a browser or device definitely fits the definition.

  Another important aspect of Google's privacy policy is its evolution. In January 2001, we can see the first traces of the collection of information as the policy was updated to include: "Google notes and saves information such as time of day, browser type, browser language, and IP address with each query. That information is used to verify our records and to provide more relevant services to users. For example, Google may use your IP address or browser language to determine which language to use when showing search results or advertisements." [146]. As written, the information is used here to tailor the response to the user's device which is exactly why they were introduced in browsers in the first place.

  In July 2004, a section called "Data collection" gives new but vague details about the goal of the collected information [147]. Notably, it indicates that "Google collects limited non-personally identifying information your browser makes available whenever you visit a website" but they use this information "to operate, develop and improve our services".

  The March 2012 update marks the foundations of Google's current privacy policy but there is a very small but notable shift in how Google uses the data [148]. While previous privacy policies indicated that the collected data was primarily for Google's own use, the update enables them to share some data with their own advertising service and their partners. Then, as indicated by following updates in March 2014 and June 2015, this sharing of data extends

to browser fingerprinting as "Cookies and anonymous identifiers" transformed into "Cookies and similar technologies" [150] with a precision that Google's partners use them [149].

- **Facebook** The Data policy indicates the collection of "Attributes such as the operating system, hardware version, device settings, file and software names and types, battery and signal strength, and device identifiers" [145]. They add that they "use the information we have to help verify accounts and activity, and to promote safety and security on and off of our Services, such as by investigating suspicious activity or violations of our terms or policies".

- **Wikipedia** The Wikimedia Foundation is very transparent on what they collect and what they do with the information. First, they make the distinction between "Information We Receive Automatically" (i.e. information from HTTP headers) and "Information We Collect" [153]. All the data related to device identification falls into the first category as it is only passive fingerprinting. They do not send a script to the browser to perform active fingerprinting. Then, they explain exactly what they do with the data: "Put simply, we use this information to enhance your experience with Wikimedia Sites. For example, we use this information to administer the sites, provide greater security, and fight vandalism; optimize mobile applications, customize content and set language preferences, test features to see what works, and improve performance; understand how users interact with the Wikimedia Sites, track and study use of various features, gain understanding about the demographics of the different Wikimedia Sites, and analyze trends.".

- **Amazon** The Amazon Privacy Notice indicates the collection of "computer and connection information such as browser type, version, and time zone setting, browser plug-in types and versions, operating system, and platform" and they add they "may also collect technical information to help us identify your device for fraud prevention and diagnostic purposes" [142].

- **Microsoft** The Microsoft Privacy Statement states they collect "data about your device and the network you use to connect to our products. It includes data about the operating systems and other software installed on your device, including product keys. It also includes IP address, device identifiers (such as the IMEI number for phones), regional and language settings" [152]. It is then unclear how these pieces of information are used as the "How We Use Personal Data" section gives global details on all the data they collect.

- **Apple** The privacy policy is very similar to the others: "As is true of most internet services, we gather some information automatically and store it in log files. This information includes Internet Protocol (IP) addresses, browser type and language, Internet service provider (ISP), referring and exit websites and applications, operating system, date/time stamp, and clickstream data" [143]. They add that they "use this information to understand and analyze trends, to administer the site, to learn about user behaviour on the site, to improve our product and services, and to gather demographic information about our user base as a whole" with a special mention that "Apple may use this information in our marketing and advertising services".

- **Yahoo** The policy is more vague in detailing the exact attributes that are collected: "receives and records information from your computer and browser, including your IP address, Yahoo cookie information, software and hardware attributes, and the page you request" [154]. They add that "Yahoo uses information for the following general purposes: to customize the advertising and content you see, fulfill your requests for products and services, improve our services, contact you, conduct research, and provide anonymous reporting for internal and external clients".

In the end, all major web actors collect device-specific information. However, it is not well-defined to which extent they collect information and what they do with it. It seems most of them rely on HTTP headers to identify the browser and the operating system but it remains to be seen if they use more active methods to fingerprint devices as the privacy policies do not indicate information collected from JavaScript or Flash.

2.2.5 Usage

Building a browser fingerprint can give many details on a device and its configuration. The use of the collected information falls into two main categories:

- **Negative or destructive use** An unknown third party would want to track a user without his consent or to attack his device by identifying a known vulnerability.
- **Positive use** Users can be warned if their device is out of date by recommending specific updates. The security of online services can also be reinforced by verifying that a device is genuine and that it is known to the system.

Tracking

As browser fingerprinting can uniquely identify a device on the web, the implications on privacy are important. By collecting browser fingerprints on several websites, a third party can recognize a user and correlates his browsing activity within and across sessions. Most importantly, the user has no control over the collection process as it is completely transparent since the tracking scripts are silent and executed in the background. The Panopticlick study outlines in more details how fingerprinting can a be a threat to web privacy [72]:

- **Fingerprints as Global Identifiers** If a device has a fingerprint that is unique, it can be identified on the web without the need of other identifiers like a cookie or an IP address. Peter Eckersley add in his study that it is "akin to a cookie that cannot be deleted". Users funnelling their network packets through a VPN (Virtual Private Network) are particularly vulnerable to browser fingerprinting as the VPN will only mask the IP address but it will not change the browser's information.
- **Fingerprint + IP address as Cookie Regenerators** Most services on the web still strongly rely on cookies to identify users as it is fast and easy to use. To protect their privacy and to prevent a third party from building a large profile of their browsing patterns, some users remove cookies and install browser extensions to manage them in a very specific way. However, coupled with a fixed IP address, a browser fingerprint can be used to regenerate deleted cookies. Researchers have already observed in the wild that any browser storage mechanisms like Flash local storage [173], HTML5 Web storage [32] or IndexedDB databases [13], can be used to "respawn" HTTP cookies. Browser fingerprinting can be added to this list but since the detection accuracy greatly vary between devices, it needs to use another source of information like the IP address to have a high probability of identifying the right device.
- **Fingerprint + IP address in the Absence of Cookies** In the absence of cookies, browser fingerprinting can be used to unmask different machines hiding behind the same IP address. In the case of company who provide the same devices to its employees, using browser fingerprinting may not prove to be successful but in a more heterogeneous environment, it is a method that can lead to conclusive results.

As a whole, browser fingerprinting can be particularly dangerous to privacy as browsers do not provide any controls over it. In Europe, the "Working Party on the Protection of Individuals with regard to the processing of Personal Data", which seeks to harmonise the application of data protection rules throughout the EU, published an opinion on device fingerprinting [79]. The documents states that websites must obtain the consent of the user to collect a browser fingerprint. To make it even clearer, they add in the document that "Tracking for online behavioural advertising" and "First-party website analytics" are affected by this directive and they also require the explicit user's consent. This approach is exactly the same as the one adopted for cookies with the EU cookie law [78] which requires users to accept the use and storage of cookies in their browsers. An update of an ePrivacy regulation has also been proposed and states that "Information related to the end-user's device may also be collected remotely for the purpose of identification and tracking" but that it "should be allowed only with the end-user's consent and for specific and transparent purposes" [80].

In the end, regulators are already in an uphill battle to verify if companies are complying with these European rules as the necessary controls cannot easily be performed online. For cookies, a simple check in the browser storage can verify if a cookie has been placed or not without the user's consent. For fingerprinting, the detection is much more complex as the browser has no mechanism dedicated to it. Regulators will need to find new ways to cooperate with companies to make sure that the privacy of web users is respected.

Identifying device vulnerabilities

A browser fingerprint is not just a simple collection of device-specific information. It truly reflects the actual set of components that are running on a device. By analysing its content, attackers can identify potential security vulnerabilities by cross-referencing the list of installed components with a database like CVE (Common Vulnerabilities and Exposures [63]). They can then design the perfect payload to target a specific device knowing its vulnerabilities in advance. For example, through the *navigator.plugins* property, one can know if a device is running an outdated version of the Flash plugin. At the time of writing, the CVE database reports 1,006 Flash vulnerabilities and more than 84% are labelled as critical, including the most recent ones [62]. If the Flash player is not up to date, users open themselves to serious security risks as any attacker on the web could execute malicious code remotely on their device.

Launching a targeted attack with the help of browser fingerprinting is not new and has been observed in the wild. Malwarebytes and GeoEdge have documented extensively with the "Operation fingerprint" how malicious advertising campaigns use fingerprinting to deliver malwares to vulnerable devices [133]. Their process is very straightforward. They hide fingerprinting code directly into the JavaScript of fake advertisers and they look from there if the device is vulnerable or not. If it is, the device will be presented with "an ad laced with malicious code that ultimately redirects to an exploit kit". If it is not, the ad will be "benign". To illustrate their findings, they detail several types of malvertising campaigns. One of them is called the *DoubleClick* campaign and it is particularly interesting as it checks for several system properties before launching an attack:

1. The IP address must be unique. This protection is put in place to prevent security researchers and security companies from replaying the attack.

2. The device must have Internet Explorer 10 or below so that they can use known vulnerabilities to execute their exploit.

3. Security products from Malwarebytes, Kaspersky, TrendMicro, Invincea as well as others must not be installed on the device.

If all these conditions are met, the device will be redirected to a page where an exploit kit is patiently waiting to be executed. Attackers take this precaution to deliver their exploits because it helps them stay under the radar for long periods of time. This is especially true as the document reports the use of techniques like encrypting the payload file or hiding malicious code into a GIF file to avoid being detected.

Another example comes from the *musical4* malvertising campaign. Figure 2.6 presents decrypted and deobfuscated JavaScript code as reported in page 10 of the "Operation fingerprint" document. Since version 5.5 in 2000, Internet Explorer dropped support of the NPAPI plugin architecture in favour of ActiveX controls. In terms of functionalities, they are identical to plugins as they provide a way to interact with the operating system from the browser. However, detecting ActiveX controls is different than just accessing the *navigator.plugins* property as it was only introduced in Internet Explorer 11 [141] To detect ActiveX controls, scripts must probe each plugin individually by creating an *ActiveXObject*. If the control exists and if the security settings allow it, the script will have access to it. If not, the browser will return the following error (tested in Internet Explorer 9): "Automation server can't create object". On line 9 of Figure 2.6, the script tries to detect if one version of the "Virtual Keyboard plugin" from a Kaspersky security solution is installed on the system. If it is not, this means that the system is probably unprotected and lines 18-19 will then redirect the browser to a malicious website where an exploit is waiting to be executed.

```
1  //Create a list of ActiveX controls to be checked
2  var document = document,
3    keyboardAPI = "Kaspersky.IeVirtualKeyboardPlugin.JavascriptApi",
4    keyboardAPIArray = [keyboardAPI, keyboardAPI + ".1", keyboardAPI + ".4_5_0.1"],
5    success = false;
6  //Check if one version of the Virtual Keyboard control is present
7  for(var i = 0; i < keyboardAPIArray.length; i++) {
8    try {
9      new ActiveXObject(keyboardAPIArray[i]);
10     success = true;
11     break;
12   } catch(e) {}
13 }
14 var redirector = "https://worldbesttraffic.eu/",
15   params = "[...]";
16 //In case no plugins are found, the script will create a malicious
17 //redirection through a malicious iframe where an exploit is waiting
18 if (!success)
19   with(document.body.appendChild(document.createElement("iframe"))) [...]
```

Figure 2.6: Excerpt from one script of the *muscial4* malvertising campaign

Finally, one could be positioned on the other side of the spectrum where browser vulnerabilities could be identified in the aim of patching them. Imagine system administrators who handle thousands of different configurations on a network. With a simple security scan, they could identify easily devices with outdated components and they could deploy fixes and updates really quickly. In 2015, Duo Security reported that 46% of corporate PCs ran outdated versions of browsers, Flash and Java [70]. In order to fix this problem, they developed a self-remediation system that warns users of out-of-date software at login, enabling them to update their own devices immediately [69]. In the end, browser fingerprinting can be used to perform targeted attacks but can also be deployed in a security solution to detect unpatched software vulnerabilities.

Improving security on the web

Fraud prevention Another use of browser fingerprinting is to improve security on the web by verifying the actual content of a fingerprint. As there are many dependencies between collected attributes, it is possible to check if a fingerprint has been tampered with or if it matches the device it is supposedly belonging to. For example, the actual platform of a device is written in clear letters in attributes like the *user-agent* or the *navigator.platform* property. However, it can also be inferred from other attributes by checking the file extensions of the list of plugins or by checking the presence of OS-specific fonts. As these attributes do not clearly indicate "Windows" or "Linux", it is much more complicated to design a solution that sends a coherent message to tracking scripts as the attributes in a fingerprint are strongly linked with each other.

ThreatMetrix, a security company that specializes in the verification of online transactions, announced in 2010 the adoption of browser fingerprinting techniques to prevent online fraud [95]. They wrote in their press release that fraudsters change their IP address, delete cookies and botnet scripts randomize device attributes. They add that relying exclusively on cookies is no longer adequate to verify an online transaction. Another security company called MaxMind also utilizes browser fingerprinting in their fraud detection services to detect fraud [94]. In that landscape, companies are turning to browser fingerprinting to be competitive in this continual arms race against fraudsters.

On the academic side, the literature on fraud detection is much thinner with only a single publication addressing this problem. Researchers at Google designed a solution called Picasso based on canvas fingerprinting to filter inorganic traffic [45]. By using specific graphical primitives from the canvas API, they are able to successfully detect the browser and OS family of a device and see if there is a mismatch between the exhibited fingerprint the actual device running the code. For example, they can distinguish between traffic sent by an authentic iPhone running Safari on iOS from an emulator or desktop client spoofing the same configuration. They add that the applications

are numerous including locking non-mobile clients from application marketplaces, detecting rogue login attempts and detecting emulated clients. Their study does not give information on the deployment of Picasso in a current Google solution but a talk at Black Hat Asia 2016 hints at its integration into Google's reCAPTCHA technology [171].

Augmented authentication At a time where passwords are the go-to solution for authentication on the web, browser fingerprinting can provide a much needed addition to reinforce the security of online accounts. By verifying the fingerprint of a device at login time, a system can easily block unauthorized access from new and unknown devices. For example, if the login system sees that an account is accessed from an iPhone while the user always connects through her Android smartphone, this change can raise some serious red flags and the server can ask for an additional confirmation before proceeding any further. Moreover, browser fingerprinting has the advantage that it is completely transparent to the user so it does not provide additional usability burdens during login time.

Alaca et al. studied extensively the use of device fingerprinting for web authentication [18]. They classify in total 29 different attributes from browser information to the network stack according to criteria like repeatability, low resource use or spoofing resistance. One important aspect considered in their study is the notion of stability. As a fingerprint is the direct reflection of what is installed on a device, a browser fingerprint constantly changes. It is then up to the login system to decide if the differences between two fingerprints are acceptable or not. For example, does a change of browser version in the user-agent come from a legitimate update of the device or from a different device altogether? If ten fonts are removed, did the user uninstall a particular software or does it come from a different device that does not belong to the user? These questions have no easy answer and each collected attribute has its own behaviour depending on the system being used or the type of the device. Spooren et al. looked at mobile devices and noticed that mobile fingerprints are predictable contrary to desktop fingerprints [175]. In Chapter 3, we provide our own results regarding the diversity of mobile fingerprints and we will see that our conclusion differ from theirs. The same authors also published an article where they investigated the use of battery information for mobiles devices in a multi-factor authentication scheme [176]. By using binary classifiers to classify battery draining and charging behaviours, they confirm that battery charge measurements can be used to contribute to an active authentication system.

Finally, some companies include in their portfolio fingerprinting solutions to augment authentication. SecurAuth is a provider of an adaptive access control solution. As part of their multi-factor authentication process, they include an heuristic-based authentication system through device fingerprinting [170]. To deal with fingerprint changes, a system administrator can set the weights of each component depending on their significance. From these weights, a score is computed and then compared to specific thresholds to see if an additional authentication factor is required or not. Another company called iovation has a solution named ClearKey [52] that integrates the collection of device information as part of their multi-factor authentication framework. They provide their own approach to deal with fingerprint changes with fuzzy logic algorithms [53].

Our contribution The use of browser fingerprinting for web authentication is still a fairly recent development. As the content of a fingerprint can change at any time, it remains key to collect the right information at the right time; otherwise, it leaves the system vulnerable to many attacks. While the literature gives an insight into the directions browser fingerprinting can take to augment authentication, it remains to be seen how everything holds when applied to a real userbase. In Chapter 5, we bring our own contribution to the domain by detailing our authentication scheme based on canvas fingerprinting. The main goal of our canvas mechanism is to strengthen the security provided by a multi-factor authentication scheme and works alongside it. With the analysis of more than a million canvas renderings, we show that our scheme provides strong guarantees in terms of security. Especially, our scheme fights off common attacks like replay or pre-play attacks thanks to its challenge-response design.

2.3 Diversity of collected information

In this section, we look at the different types of information that can be collected through a web browser. We report on the existing literature on browser fingerprinting as well as other techniques

found in scripts on the Internet. The primary goal of this section is to show that the collected attributes in a fingerprint can be very diverse and span all layers of the system from the browser to the operating system to the hardware. Appendix A provides snippets of code that explain how most of the attributes are collected. Fingerprinting scripts do not use any invasive techniques as they rely on publicly available APIs to build a fingerprint. For most attributes, only very few lines of code are needed to be collected without the need to enter complex programming procedures.

It should be noted that the attributes listed here represent a snapshot in time of what is possible to collect through browser fingerprinting. As new APIs are introduced and as browser vendors tweak the behaviours of current APIs, attributes that can be collected today (i.e. at the time of writing of this thesis) can be vastly different from the ones that will be collected in the future. Chapter 3 reports on the evolution of the browser fingerprinting landscape in the last 5 years. The discovery of new fingerprinting techniques coupled with pushes in new directions from major web actors make browser fingerprinting a fascinating and fast-evolving territory that can be full of surprises.

2.3.1 HTTP headers

An HTTP request is composed of header fields to define the operating parameters of an HTTP transaction. Section 14 of RFC 2616 [166] defines the syntax and semantics of all standard HTTP header fields. Below, we detail the headers that can be collected to build a browser fingerprint and we identify discriminating information.

- **User-agent**: Its goal is to inform the server about client limitations so that the response is tailored to the user's device. It contains information about the browser and its version along with details on the operating system. The actual structure of the header comes from a rich history which is detailed in Section 2.1.1.

```
//Examples
Mozilla/5.0 (X11; Fedora; Linux x86\_64; rv:52.0) Gecko/20100101 Firefox/52.0
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
    Chrome/56.0.2924.87 Safari/537.36
Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_3) AppleWebKit/602.4.8 (KHTML,
    like Gecko) Version/10.0.3 Safari/602.4.8
```

- **Accept**: As defined by RFC 2616, this header "can be used to specify certain media types which are acceptable for the response". Nowadays, as most browsers are supporting the same set of standards, this header has lost some of its purpose since devices present much less limitations than in the past. However, we can still notice some differences as some browsers are standing out from others by natively supporting new non-standard formats like Chrome with the "WebP" image format [201] or Edge with the "JPEG XR" image format [104].

```
//Examples
//Firefox
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
//Chrome
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
//Edge
text/html,application/xhtml+xml,image/jxr,*/*
```

- **Accept-Encoding**: It indicates the encoding or compression formats supported by the browser. Most web pages are compressed with the "gzip" format but some browsers support more advanced compression formats such as Brotli [161].

```
//Examples
gzip, deflate, br //from Firefox
gzip, deflate, sdch, br //from Chrome
gzip, deflate //from Safari
```

- **Accept-Language**: It informs the server of the set of languages that are preferred for the response. The languages are organized in decreasing order of priority. In the examples, the values after the "q" letters indicate the exact value of the priority.

```
//Examples
en-US,en;q=0.5
fr-FR,fr;q=0.8,en-US;q=0.6,en;q=0.4
en-us
```

- **Order of HTTP headers**: When a browser performs an HTTP request, it sends all its HTTP headers in a specific order. Unger et al. noted in one study that there is a significant difference in how the browsers order the HTTP header fields [193].

2.3.2 JavaScript

We list here all the information that can be collected by the execution of a simple JavaScript script in a browser. All the attributes presented here do not require any special permissions to be accessed. Any server on the web can build a fingerprint with this information even though some may require more time to be collected than others (i.e. benchmarking or probing browser features require more time than accessing a simple object property).

- **List of plugins**: Through the *navigator.plugins* property, a script can get access to the list of installed plugins. It should be noted that a plugin must not be mistaken with a browser extension. While a browser extension offers ways to modify the behaviour of the browser, it is limited to a set of specific browser APIs. On the other end, plugins extend the capabilities of a browser by providing a direct link with the operating system. This enables the browser to support formats that are not natively supported by the browser like Flash or Silverlight and to communicate with external programs like an anti-virus. However, as we will see in Chapter 3, the old plugin architecture called *NPAPI* has been deprecated from modern browsers as HTML5 now offers many features that reduce greatly the need for external plugins. Only Flash is still supported but it is a matter of time before it gets dropped by major browser vendors.

```
//Examples
//Firefox on a Linux system
Plugin 0: Shockwave Flash; Shockwave Flash 24.0 r0; libflashplayer.so.
//Chrome on a Windows system
Plugin 0: Chrome PDF Viewer; ; mhjfbmdgcfjbbpaeojofohoefgiehjai. Plugin 1:
    Chrome PDF Viewer; Portable Document Format; internal-pdf-viewer. Plugin 2:
     Native Client; ; internal-nacl-plugin. Plugin 3: Shockwave Flash;
    Shockwave Flash 24.0 r0; pepflashplayer.dll. Plugin 4: Widevine Content
    Decryption Module; Enables Widevine licenses for playback of HTML
    audiovideo content. version: 1.4.8.962; widevinecdmadapter.dll.
//Safari on a Mac system
Plugin 0: WebKit built-in PDF; ; .
```

- **Platform**: The *navigator.platform* property returns a string representing the platform on which the browser is running. This information echoes what is already present in the User-agent header.

```
//Examples
Linux x86_64
Win32
MacIntel
```

- **Cookies enabled**: The *navigator.cookieEnabled* property returns a boolean value indicating whether cookies are enabled or not.

```
//Examples
true
false
```

- **Do Not Track**: The *navigator.doNotTrack* property returns the user's Do Not Track settings. If users wish not to be tracked, they can activate Do Not Track so that a DNT header is sent along with all the browser's requests' [174]. However, since its inception, the DNT header has faced many problems as there was no incentive for companies to honor it [194].

```
//Examples
1
0
null
```

- **Timezone**: The following instruction returns the time zone difference in minutes from current locale (host system settings) to UTC: *new Date().getTimezoneOffset()*.

```
//Examples
-120 //GMT+2
-60  //GMT+1
480  //GMT-8
```

- **Screen resolution and color depth**: The *window* object provides a *screen* object which provides information on the window that is currently being rendered. Notably, the *screen* object offers the *width*, *height* and *colorDepth* properties that reflects actual characteristics of the screen that is being used by the device. However, it should be noted that the dimensions reported from JavaScript are limited to a single screen contrary to the Flash plugin which can detect a multi-monitor setup.

```
//Examples
1920x1200x24
1920x1080x24
1920x1200x24
```

- **Local and session storage**: The Web Storage API provides mechanisms to store data for a particular domain. The difference between local and session storage is that data stored in local storage has no expiration date. The user can disable this API at any time in the browser settings but most users have it activated as it is the default option. To test it, a script can try to store a value in either *window.localStorage* or *window.sessionStorage* and see if the value persists.

```
//Examples
yes
no
```

- **Use of an ad blocker**: The most popular browser extensions for both Chrome and Firefox are ad blockers with numbers easily exceeding millions of downloads. As explained in greater details in Section 2.4.3, most blocking extensions use a blacklist system to prevent a browser from downloading and executing advertising scripts. If the URL or the name of a script is present in the extension database, the script is blocked. In order to test if a device has an ad blocker installed, we look to see if a script named "ad.js" which is present in default blacklists is executed.

```
//Examples
yes
no
```

- **Canvas (2D)**: As defined by the specification of the W3C [28], the "2D Context provides objects, methods, and properties to draw and manipulate graphics on a canvas drawing surface". Users can draw and animate any number of shapes and they can render textual content directly in the browser by using the graphical capabilities of the device. With regards to fingerprinting, any script can create an HTML canvas element and interact with it through a *CanvasRenderingContext2D* object. When an image has been rendered, the *getDataUrl()*

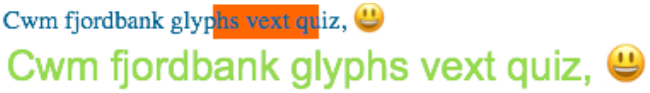
function can be called to get a string representation of the image, thus allowing easy comparisons between devices. It should be noted that an image can be collected without ever being presented to the user as a canvas element can stay invisible for the whole duration of the rendering process.



(a) Rendering from a device running Firefox on Fedora 25



(b) Rendering from a device running Chrome on Windows 7



(c) Rendering from a device running Safari on Mac OS X

Figure 2.7: Examples of canvas renderings on three different devices

In 2012, Mowery and Shacham were the first to study the canvas API and the canvas 2D context in their Pixel Perfect study [116] to produce fingerprints. To expose differences between devices, they draw several lines of text with different fonts in a canvas element. One line uses the *Arial* font which is commonly found in modern operating systems. From 300 canvas samples, they observed 50 distinct renderings with the largest cluster containing 172 samples. As the font handling stacks can vary between devices, they state that the operating system, browser version, graphics card, installed fonts, sub-pixel hinting, and antialiasing all play a part in generating the final user-visible bitmap. This is especially true as the file providing the Arial font may slightly differ between operating system and distribution. To remedy this problem, they try with a WebFont which is a font directly provided by the server. Differences observed from that test are then a direct consequence of the font engine used to render the text and not from the font file itself. From 294 samples, they observe similar results as the Arial test with 45 distinct results. They hint in the study at a more aggressive way to test devices with what they call "nonsense text". Instead of asking for a font that exists, they make an invalid font request to rely on the fallback font mechanism of the browser. As the browser cannot find the requested font, it will used the default font that is indicated in its settings. Depending on the operating system, this fallback font can greatly vary creating even more differences between devices. However, there is no mention in their study of an experiment with this "nonsense text".

Acar et al. performed a large-scale study of canvas fingerprinting in "The Web Never Forgets" [13]. They crawled the Alexa most popular 100,000 websites to find canvas fingerprinting scripts on the web. By instrumenting the browser, they are able to record calls to known functions of the canvas API and decide if a script performs canvas fingerprinting or not. They found that 5.5% of visited websites had a canvas script on their homepage even though 95% of the scripts belong to a single provider called *addthis.com*. Following this study and the backlash that ensued, *addthis.com* responded that canvas fingerprinting was used for an R&D test and that the code has been disabled since [17]. One of the most important aspects of Acar et al. study is their analysis of the canvas fingerprinting scripts. They found that scripts utilize the techniques outlined by Mowery and Shacham and notably, they take advantage of the fallback font mechanism to generate even more differences between devices. This is the first time that such techniques were reported on the Internet. They also noticed that most scripts share a very similar codebase and they explained this similarity by the availability on GitHub of an open source fingerprinting library called *fingerprintjs* [87].

Finally, the latest numbers on the adoption of canvas fingerprinting on the web come from

a study conducted by Englehardt et al. in 2016 with the OpenWPM framework [77]. They show that 1.6% of the Alexa top million websites have a canvas fingerprinting script and they confirm that *addthis.com* has stopped using this technique as they did not find any scripts from this domain.

- **WebGL (3D)**: As designed by the Khronos Group [30], WebGL is a graphics API that can render interactive 3D objects in the browser and manipulate them through JavaScript without the need for plugins. It relies on the browser canvas element but instead of using a *CanvasRenderingContext2D*, it has its own *WebGLRenderingContext*. If a graphic card or GPU is available, the browser can make use of it to render complex 3D scenes. If hardware rendering is unavailable, the browser can fallback on the CPU with a software renderer like SwiftShader that Chrome uses [178].

  Mowery and Shacham also studied in Pixel Perfect [116] the use of WebGL for fingerprinting. In their test, they created a 3D surface on which they apply a very specific image and they add different ambient lights. They observed 50 distinct renders from 270 samples. They explain this heterogeneity by the difference in hardware and software where the processing pipeline is not exactly identical between devices. However, it is not until 2017 that progress was made with regards to the capabilities of WebGL for fingerprinting. Cao et al. designed a fingerprinting technique that relies heavily on WebGL to identify devices [48] (more info on their results in Section 2.4.1). Through a series of 31 rendering tasks, they test carefully selected computer graphics parameters to extract device features and they were able to uniquely identify more than 99% of 1,903 tested devices.

- **WebGL Vendor and renderer**: The WebGL API supports extensions to offer additional features for developers. The current list of extensions can be found in the official Khronos Extension Registry [198]. One of these extensions is called "WEBGL_debug_renderer_info" and provides information on the actual GPU vendor and renderer. As the name indicates, it should be used for debugging purposes but that does not prevent a fingerprinting script from accessing these values. The actual values of the vendor and renderer reflect the environment in which WebGL is executed. It can report the exact value of the GPU used by the device or it can give the name of the software renderer used in case the browser falls back to the CPU. It should be noted that not all browsers give access to the WebGL Vendor and Renderer. While any script can query these values on Chrome, Firefox requires a privileged context activated by a browser flag to unmask the real values (it returns "Mozilla" in an unprivileged context).

```
//Examples
//WebGL Vendor
Intel Open Source Technology Center
Google Inc.
ATI Technologies Inc.
NVIDIA Corporation

//WebGL Renderer
Mesa DRI Intel(R) Haswell Mobile
Google SwiftShader //Shows the use of a CPU and not of a dedicated GPU
AMD Radeon R9 M370X OpenGL Engine
GeForce GTX 970/PCIe/SSE2
```

- **AudioContext**: Discovered by Englehardt et al. while crawling the web looking for trackers, AudioContext fingerprinting is one of the latest additions in a fingerprinter's toolbox [77]. The AudioContext API provides an interface to create a pipeline to process audio. By linking audio modules together, you can generate audio signals and apply very specific operations like compression or filtering to generate a very specific output. They found scripts that process an audio signal generated with an *OscillatorNode* to fingerprint the device. The authors add that the fingerprinting process is similar to what is done with canvas fingerprinting as processed signals will present differences due to the software and hardware stack of the device. The relative novelty of this technique explains that scripts using this API were only found on a very small number of websites.

- **JavaScript standards conformance**: Muzanni et al. proposed a method to reliably identify a browser based on the underlying JavaScript engine [121]. They analysed browsers to see if they complied with the JavaScript standard and they tested them to detect which features were supported. By collecting a dataset from more than 150 browser and operating system combinations, they were able to compute the minimal suite of tests that needs to be run to identify uniquely each combination. Their approach is possible because web browsers present differences in the JavaScript engine even between two subsequent versions.

- **CSS querying**: Unger et al. perform a series of test to detect CSS properties that are unique to some browsers [193]. For example, Firefox presents CSS properties prefixed with "-moz-" [60] while Chrome and Safari have some properties prefixed with "-webkit-" [61]. With their method, they can easily detect the browser family as these prefixes are not shared between browsers.

- **Font metrics**: Fifield et al. looked into the analysis of character glyphs to identify devices on the web [86]. They noticed that the same character with the exact same style may be rendered with different bounding boxes depending on the browser and the device used. By testing 125,000 different Unicode characters on more than 1,000 web browsers, they were able to uniquely identify 34% of their population. With the data they collected, they were able to reduce the number of tested characters to 43 to reach the same conclusion.

- **Browser extensions**: Detecting a browser extension is challenging as there is no API to query to get the exact list of installed extensions in the browser. We explained previously how it is possible to identify the presence of an ad blocker but it is much more difficult to generalize this approach for a lot of extensions. During the writing of this thesis, two studies were published regarding the detection of browser extensions.

  The first was conducted by Sjösten et al. and looked at the use of web accessible resources to detect extensions [172]. By accessing very specific URLs, they can know if an extension is installed or not. For example, to display the logo of an extension, the browser knows where it is stored on the device and it follows a URL of the form "extension://<extensionID>/<pathToFile>" to fetch it. However, since these resources can be accessed in the context of any web page, this mechanism can be abused by a script to detect the presence or absence of a particular extension. Moreover, this system is particularly precise as no two extensions share the same ID. The only way to prevent detection is not to use any web resources. In their study, they were able to detect 12,154 Chrome extensions out of 43,429 and 1,003 Firefox ones out of 14,896. The differences in numbers between Chrome and Firefox is explained by the difference in addon architecture between the two browsers.

  The second study was done by Starov et al. and consists in identifying side effects produced by extensions [177]. For example, if an extension adds a button on YouTube to provide new controls over a video, the added button is detectable by analysing the DOM of the web page (the Document Object Model represents the structure of a page). Detecting an ad blocker is similar as the blocking of an ad script will prevent some ads from being displayed. If an extension modifies any element of the page that the user is visiting, this behaviour is detectable and can lead to the identification of the installed extension. The authors of the study performed an analysis of the 10,000 most popular Chrome extensions. They tested individually every single one of these 10,000 extensions to record in a database detectable changes. They found that 9% of them produce DOM modifications that can be detected on any domain and 16.6% introduce detectable changes on popular domains.

- **Battery**: Drafted as early as 2011 [34], the "Battery Status" specification defines "an API that provides information about the battery status of the hosting device" [36]. The API is composed of a *BatteryManager* interface that reports if the device is charging or not. It also includes extra information like the charge level of the device along with its remaining charging and discharging time. As detailed by the W3C, by giving knowledge of the battery status to web developers, they "are able to craft web content and applications which are power-efficient, thereby leading to improved user experience". For example, if a portable device is running low on battery, an application could decide to check for updates less often

or simply stop sending data to preserve the battery as much as possible. The intent behind the addition of this API seems entirely legitimate.

However, they underestimated how much information regarding the battery could be misused in the wild. In 2015, Olejnik et al. performed a privacy analysis of the Battery Status API [130]. They highlighted the fact that the level of the battery could be used as a short-term identifiers across websites and that repeated readouts could help determine the capacity of the battery. In 2016, Uber disclosed the fact that users are likely to pay more if their phone is running out of battery because they do not take the time to look for different prices [191]. Even though Uber added that the company did not use the battery level to set prices for a ride, this piece of information raised of a lot of concerns about how the Battery Status API could be misused. The persons responsible for the standard did not anticipate all these problems as they only indicated in their original draft that the "the information disclosed has minimal impact on privacy or fingerprinting" [35].

To address the issues raised by this API, different approaches have been adopted. Mozilla decided to remove access to the API by any untrusted code [117]. This is an unprecedented decision in the history of the Web as this is the first time that public access to an API has been removed due to privacy concerns. This is even more impressive considering that the Battery Status API has been enabled by default in Firefox since version 11 released in March 2012 [118]. Firefox restricted the access to privileged code (access is allowed internally and from extensions) to prevent misuse of the battery information in March 2017 with the release of Firefox 52 [119]. On their end, the WebKit team has completely removed the Battery Status API from their code [199]. The team behind the Yandex browser has chosen an opt-in approach where default information is spoofed if the user has not given his consent [207].

In the end, there is no unique consensus on what should be done with this API as different browser vendors have chosen different approaches. The future is still unclear as some browsers like Chrome still supports it. However, it is a certainty that the tumultuous changes of the Battery Status API will have an impact on the design of future web standards going forward as privacy will be put at the forefront of discussions. Olejnik et al. documented extensively the complete history of the Battery Status API in [131]. Notably, they detail a list of recommendations to strongly improve the privacy engineering of web standard and to detect early on privacy problems at the specification stage.

```
//Example
Charging: Discharging
Time to charge: Infinity Seconds
Time to discharge: 8917 Seconds
Battery Level: 91%
```

- **Benchmarking**: Another way to uncover information about a device is to benchmark its CPU and GPU capabilities. Through JavaScript, a script can launch a series of tasks and measures the time it takes to complete them. However, the biggest difficulty when using benchmarks is to interpret differences and fluctuations correctly. Two time values can be different because they have been collected from two different devices but they could also belong to a single device where a new background process came disrupting the actual measurements.

  Mowery et al. were the first to study how benchmarking could be used to fingerprint a device [115]. They use 39 different tests to identify the performance signature of the browser's JavaScript engine. They show that they are able to detect the browser and its version with a 79.8% accuracy. However, the biggest downside of their approach is that it takes in total 190.8 seconds to run the complete benchmark suite. Contrary to the majority of the attributes presented in this section that can be collected in a matter of milliseconds, this time difference makes it almost impossible to deploy such methods in the wild. They also add in their study that they can build a more complete fingerprint by identifying the underlying OS and the CPU architecture but it requires a very extensive dataset to detect these properties accurately.

  Nakibly et al. turned to the GPU to benchmark devices [122]. They use the WebGL API to display complex 3D scenes and they measure the number of frames rendered by the browser.

While their experiment is small in scope, they show that benchmarking the GPU can produce very noticeable differences between devices as a small GPU on a smartphone will behave very differently than the latest high-end graphic card.

Finally, Saito et al. tried estimating the presence of specific CPU features through benchmarking [168]. From a dataset composed of 350 devices, they are able to detect with 99% accuracy the presence of AES-NI which is a special instruction that provides hardware acceleration for AES processing. They can also measure to some extent if TurboBoost is activated with an accuracy of 84.7%. It is a technology that increases the frequency of the CPU in case of heavy load. However, they still face the same shortcomings as the other studies because the benchmarks for AES-NI and TurboBoost take respectively 28 and 46 seconds to be fully executed. These long time durations prevent these methods from actually being used in real-world applications.

2.3.3 Plugins

Plugins are a part of the web since the beginning. In 1995, the Netscape Navigator introduced the Netscape Plugin Application Programming Interface (NPAPI) to add support for plugins. The intent behind it was to make the browsing experience richer by supporting new formats that the browser did not handle natively. Through the years, many plugins were developed like Flash, Silverlight, Java, Shockwave or Acrobat Viewer and they made the web more dynamic. However, as we will see in Chapter 3, plugins are in their last breath as they proved to be the source of too many security and stability problems. Flash is still supported but other plugins cannot run anymore on modern browsers as they dropped the support of the NPAPI legacy architecture.

Plugins are interesting for fingerprinting as they do not have the same limitations as a browser API. Since they rely on software that is directly installed on the operating system, they can have access to very precise information on the device. In this section, we look at what can be retrieved through the Flash plugin as it complements very well what can already be collected through JavaScript. Notably, we look at the *Capabilities* class which "provides properties that describe the system and runtime that are hosting the application" [91]. The code listed for each attribute is written in ActionScript [15] and the Flash functions are called directly from JavaScript with the help of the *ExternalInterface* class [92];

- **List of fonts**: Flash gives access to the exact list of fonts installed on the operating system with a simple call to the *Font.enumerateFonts* function. This method is fast and direct compared to the JavaScript alternative that consists in probing each font one by one. In 2013, the FPDetective study by Acar et al. found 95 out of the top 10,000 websites performing Flash-based font enumeration and 404 sites out of the top million websites doing JavaScript-based font probing [14]. Saito et al. also showed that it is possible to identify the operating system along with some installed applications just by analyzing the list of fonts [167]. This is made possible because some OS and applications come with their own set of fonts that they are the only one to provide.

```
//Examples
Abyssinica SIL, Aharoni CLM, AR PL UMing CN, AR PL UMing HK, AR PL UMing TW,
    AR PL UMing TW MBE, Bitstream Charter, Bitstream Vera Sans, Bitstream Vera
    Sans Mono, Bitstream Vera Serif, Caladea, Caladings CLM, Cantarell...
Agency FB, Aharoni, Algerian, Andalus, Angsana New, AngsanaUPC, Aparajita,
    Arabic Typesetting, Arial, Arial Black, Arial Narrow, Arial Rounded MT Bold
    , Arial Unicode MS, Baskerville Old Face, Batang, BatangChe, Bauhaus 93...
```

- **Screen resolution**: Getting the screen resolution through Flash may appear superficial as it can already been collected through JavaScript. However, Flash stands out by reporting the full screen resolution of a multi-monitor configuration instead of the resolution of a single screen.

```
//Examples
3840x1200
1920x1080
5760x1080
```

- **Language**: As defined by the documentation, *Capabilities.language* specifies the language of the operating system on which the browser is running. For most users, collecting this attribute may not provide new information as their browser will reflect the language of their desktop environment. But for others, it may provide just enough information to identify them if they tweaked their browser to use a language that is different than the default one of their OS.

```
//Examples
en
fr
es
```

- **Platform**: The *Capabilities.os* property reports the current operating system. However, contrary to the information collected from JavaScript, it can be much more precise. On Mac or Windows, Flash indicates the exact version of the OS. And for Linux systems, Flash reports the exact version of the Linux kernel. This divides the entire Linux population into different groups because the way kernel updates are delivered can vastly vary between Linux distributions. The presence of kernel information can also pose a serious security risk as it can be a vector for a targeted attack if vulnerabilities are known for the indicated kernel version.

```
//Examples
Linux 4.10.5-200.fc25.x86\_64 //'fc25' is for Fedora 25
Linux 4.8.7-1-ARCH  //'ARCH' is for the Arch Linux distribution
Windows 7
Mac OS 10.9.5
Samsung TV Linux 3.8.13
```

2.3.4 Conclusion

In this section, we showed that a lot of diverse information can be collected from publicly available browser APIs: from the browser, its version and its preferences to the operating system to the actual GPU and CPU that are running on the device. However, it should be pointed out that not all attributes give the same level of details. As we can see with the DNT header or the presence of cookies, some attributes have very few values like *yes* or *no* that are shared by many devices. Others can be very discriminating as they can report the exact model of the GPU or the exact version of the Linux kernel. This is why it is preferred to collect as much information as possible to perform identification as it is impossible to predict the level of precision from the collected data beforehand. Gulyás et al. explored the concept of near-optimal fingerprinting with constraints, i.e. fingerprinting in a context where only a limited number of the user's attributes can be queried [97]. They showed that, from a dataset of 43,656 lists of fonts, they were able to uniquely identify 30.58% of users just by checking 5 fonts for each of them. This proves that most attributes may not be needed when identifying a device in a dataset.

Another important notion is stability. As users update their devices and customize their browsers, the collected fingerprints are changing. In order to link browser fingerprints belonging to the same device, the quantity of modified information should be as small as possible. For example, a server can expect to see an increase in the browser version from time to time but a change in the timezone is more unusual as it implies that the user is travelling. Different behaviours and different types of changes must be identified to be able to identify a device with accuracy as each attribute evolves in its own way.

2.4 Defences against fingerprinting

In this section, we detail techniques and solutions aimed at mitigating the effects of browser fingerprinting. The goal is to improve users' privacy by preventing unwanted tracking. As we will see, there is no ultimate approach that can prevent fingerprinting while keeping the richness of a modern web browser. Designing a strong defence requires a fine-tuned balance between privacy and usability that can be challenging to get right.

2.4.1 Increasing device diversity

Modifying the content of fingerprints

The first defence to mitigate browser fingerprinting is to increase the diversity of devices so that real fingerprints are hidden in noise. The intuition behind this method is that third parties rely on fingerprint stability to link fingerprints to a single device. By sending randomized or pre-defined values instead of the real ones, the collected fingerprints are so different and unstable that a tracking company is unable to identify devices on the web.

While this approach can appear to be strong on paper, the reality is much more complicated as Peter Eckersley called it *the Paradox of Fingerprintable Privacy Enhancing Technologies* [72]. Instead of enhancing users' privacy, some tools make fingerprinting easier by rendering a fingerprint more distinctive. By looking through the extensions available for both Chrome and Firefox, one can find many *spoofers* or *switchers* to modify the actual values that are collected by scripts. One of the most popular one on Firefox called Random Agent Spoofer [157] claims more than 200,000 users and it provides the ability to rotate "complete browser profiles (from real browsers / devices) at a user defined time interval". Nikiforakis et al. performed an analysis of these extensions and found many issues with regards to browser fingerprinting [124]. They showed that they might be harmful as they did not cover all the possible ways of discovering the true identity of a browser. First, these extensions provide incomplete coverage of modified objects. The researchers noted that one extension was modifying the *navigator.userAgent* property but forgot to change others property like the *platform* or the *appName* This poses a problem as not only a script can detect that a protection is running but the lack of coverage offers the opportunity to uncover the real value that the extension was trying to hide. Second, some extensions create impossible configurations where there is a mismatch between different attributes. One browser could announce in its user-agent that the underlying OS is Linux while the navigator.platform property indicates it is running on Windows. Another example would be a device claiming to be an iPhone while the reported screen resolution is far bigger than what is currently supported on these devices. Third, while most attributes are collected through JavaScript, a script can rely on HTTP headers or plugins like Flash to collect extra information. Nikiforakis et al. witnessed that some extensions were changing the HTTP headers but forgot to change the same values accessible through JavaScript. In the end, while the idea of switching values with other ones is promising, the constant evolution of browsers coupled with very strong links between attributes prevent this approach from being recommended. To fix the shortcomings of these agent spoofers, the scientific community turned itself to new approaches that are outlined below.

Torres et al. explored the concept of separation of web identities with a solution called FP-Block [185]. When the browser connects to a new domain, it will generate a new identity (i.e. a new fingerprint) for this particular domain. If at some point the browser encounters the same domain again, it will return the fingerprint associated with it. The intuition behind FP-Block is that third parties will see different fingerprints on each site they are embedded so that tracking is hampered. However, their approach presents the same limitations as naive spoofers since the modified values are incomplete and can be incoherent.

FaizKhademi et al. developed the FPGuard solution which runs in two phases: detection and prevention [83]. First, they detect fingerprinting-related activities with a series of 9 metrics. For example, they check the number of access to the *navigator* and *screen* objects and they count the number of fonts loaded using JavaScript. From these metrics, they compute a suspicion score and if this score goes above a specific threshold, the script will be considered as a fingerprinting one. From there, the second phase kicks in. FPGuard can launch a series of components that will modify the content of a fingerprint. They target very specific attributes, namely the *navigator* and *screen* objects, the list of plugins, the list of fonts and canvas renderings. They designed special randomization policies that dictate how each of these attributes are modified. They write for example that they change the subversion of the browser in the user-agent while fake plugins are added in the plugins list. Unfortunately, the main problem with the FPGuard study is the lack of information given in the article. There are no details on how the thresholds are computed in the detection phase and the authors never explain the exact modifications performed in the prevention phase. For this reason, it is not possible to discuss in greater details the advantages or weaknesses

of this approach.

Fiore et al. worked to counter unwanted tracking by creating fingerprints that resemble the ones left by someone else [88]. They claim that they have to alter data in a way that is consistent to prevent being detected. However, their solution implemented in Google Chrome present the same shortcomings as others. They modify a very specific subset of fingerprintable attributes with a fake browsing profile but they do not provide a complete coverage of them. An inconsistency can even be detected in Figure 2 of their article as there is a mismatch between the device announced as an iPad in the user-agent and the screen resolution of 800x600 which should be 768x1024.

Baumann et al. designed a solution to disguise the Chromium browser called DCB (Disguised Chromium Browser) by changing the following parameters: the screen resolution, the browser language, the user-agent, the time and date, the list of fonts and the list of plugins [37]. When DCB launches, it contacts the main server that "maintains a database of real world fingerprinting features to enforce a robust browser configuration on the client". One of the two following strategies is then applied:

- N:1 Many Browsers, One Configuration - Several devices connected to the central server will adopt the same fingerprint for a given period of time. This way, a third party will be unable to uniquely identify a device as they will all look the same to a fingerprinter. Devices are organized in groups of configurations that share similar values to avoid contradiction in exhibited fingerprints.

- 1:N One Browser, Many Configurations - The browser configuration is changed on each start of DCB, hiding at the same time the actual browser and system configuration.

DCB presents three advantages compared to other solutions. The first comes from the frequency of the changes. Most protection solutions would return different values for each execution of a fingerprint script. This introduces a problem of detectability as repeating the same test twice would yield different results. Here, DCB is designed so that it will modify values the exact same way during an entire browsing session. The second advantage is the N:1 configuration strategy. Several solutions generate fingerprints that are different from the actual device fingerprint. The problem is that the user can still be tracked in his session as he will be the only one with this generated fingerprint. The N:1 strategy removes this problem entirely as several devices present the same identical fingerprint at a given moment in time. The final advantage lies in how the configuration groups are created. Instead of putting devices with different operating systems in the same group, they try to regroup devices where the core configuration is similar (i.e. same OS and same language). As browsers are massive pieces of software, the actual implementations of each browser on different operating systems present differences that are detectable through JavaScript. By regrouping devices with the same OS, DCB prevents the introduction of OS mismatches in exhibited fingerprints.

Nikiforakis et al. explores with PriVaricator the use of randomization to render browser fingerprints unreliable for tracking [123]. The main insight of their study is that making fingerprints non-deterministic on multiple visits make them hard to link across browsing sessions. One way to break this determinism is to take advantage of randomization but the key challenge is to determine how and when to use it. As indicated by the authors, blatant lying is not such a good idea as it can negatively impact the user experience. Indeed, key browser properties like the screen resolution or the user agent are legitimately used by websites to optimize their displays. Modifying them randomly can seriously degrade the user experience as web pages could become broken and calls to missing APIs could be performed. In order to offer an effective fingerprinting protection tool that minimizes site breakage, they introduce the concept of randomization policies. Each policy details the modifications made to a specific attribute along with a set of requirements that define when it kicks in. This way, any developer can define its own modification strategy that balances effectiveness with usability. For example, as a proof of concept, they modified the Chromium source code to add different randomization policies: three against the offset measurements of HTML elements and four against the enumeration of the list of plugins. For the offset measurements, the browser starts lying after it has passed a specific lying threshold which represents the number of accesses to the *offsetWidth* and *offsetHeight* properties. Depending on the policy, the browser then returns zero, a random number between 0 and 100 or it can add 5% noise to the given results. For plugins,

a policy defines the probability of hiding each individual entry in a given list. They complete their study by testing known fingerprinting scripts with each of their policies and they crawl the Alexa top 1,000 websites several times to assess site breakage. The main difficulty with the approach chosen by PriVaricator is linked to the incomplete coverage problem mentioned earlier in this section. As browsers are getting bigger and enriched with new APIs, new ways to get access to supposedly hidden information are surfacing, rendering the proposed mitigation useless. For example, PriVaricator lies about HTML offset measurements to prevent JavaScript font probing. However, the Canvas API now offers a measureText method as indicated by [77] that gives the exact same information that the browser was trying to hide with the modified functions. If the policy is not updated to take into account this new function, the mitigation is rendered useless. Thus, maintaining a complete set of policies requires a constant watch for new APIs and changes to existing ones.

Finally, while most attributes are collected in string form, other ones from the Canvas or AudioContext APIs produce more complex data structures. This brings us to the last approach of this section. Instead of simply replacing an output with another pre-defined one, one can introduce noise into the rendering process of these APIs. This way, a Canvas or AudioContext test can be ever so slightly different at each execution. One way to introduce noise is to position the modification at the very end of the processing pipeline where a script collect its values. An extension called Canvas Defender on Firefox does exactly this [47]. When a script renders an image, the browser will behave normally and the user will see the intended image. However, when the script tries to read the content of the rendered canvas element, it will go through a function that modifies the actual RGB values of each pixel. The image collected by the script is then different from the image that the user can see. Another strength of Canvas Defender is that it addresses a problem raised in the original study on canvas fingerprinting. Mowery et al. wrote that tools that add noise are not a feasible defense against current scripts because the noise can be lifted by repeating a test a few times and comparing the results [116]. With Canvas Defender, the modifications are consistent across the same browsing session. When the browser starts, the extension generates a randomization seed that is then used throughout the entire browsing session to provide consistency across tests. Baumann et al. positioned themselves much earlier in the rendering pipeline by directly modifying the Chromium source code in DCB [37]. They modified the *fillText()* and *strokeText()* that are heavily used in canvas fingerprinting scripts to alter the renderings of canvas elements at runtime. Their approach also provides consistency in the same browsing session as they use a random session identifier generated at startup to steer the modifications.

In the end, it is possible to increase the diversity of exposed fingerprints and modify their content but the challenges to have a working and undetectable solution are numerous. Attributes cannot be modified in a way that will break browsing. The slightest mismatch between two attributes can make a user more visible to trackers which defeats the entire purpose of running a defence solution. All the techniques detailed in this section pose the question if such kind of approach should be explored further or if the constant evolution of web browsers render current implementations incredibly hard to maintain and to recommend. While researchers and developers are finding many ways to make fingerprints unstable, there are always really small details that are easy to overlook that make current solutions ineffective. Modern web browsers are such complex pieces of machinery that it is incredibly hard to predict where the next piece of revealing information will be. And at the pace at which the web evolves, it is anyone's guess if changes can be expected on that front in the near future.

Changing browsers

Since a large part of a device fingerprint is composed of browser-specific information, one could decide to use two different browsers to have two distinct device fingerprints. This way, it is harder for a third party to have a complete picture of a user's browsing patterns as the tracking party will obtain two decorrelated browsing profiles. While the premise behind this idea is really simple, the truth behind it is more complicated. Two studies have shown that collecting attributes that are specific to the OS and the hardware can be sufficient to uniquely identify a device.

Boda et al. designed a browser-independent fingerprinting algorithm that rely mainly on attributes like the list of fonts, the timezone and the screen resolution [41]. Their findings show that

the list of fonts provide a solid base for identification and that they were able to identify returning visitors who used more than one browser or changed their IP addresses dynamically. However, the small size of their dataset prevented them from drawing more general conclusions as the collected data was not diverse enough to see if it holds at a larger scale.

Cao et al. designed a fingerprinting technique that relies heavily on the OS and hardware functionalities of a device [48]. By rendering 31 different tasks with the WebGL API, they are able to extract device features from carefully selected computer graphics tests. They also collect the size of the screen along with the list of fonts of the device to complement their dataset. By collecting data from 1,903 participants, they show that they are able to uniquely identify more than 99% of devices even if the user switches browser. One important detail is that their whole suite of tests take several seconds to be fully executed contrary to more standard fingerprinting scripts which take less than a second. A run of the complete suite from their Unique Machine demo website [49] takes about 15 seconds and as some tasks are aggressive on both the GPU and CPU, the browser became unresponsive for several seconds during the test.

In the end, cross-browser fingerprinting is a reality even if its deployment in a real-world solution may prove very challenging mainly due to time constraints. By collecting enough data from the OS and hardware layers of a system, a third party can uniquely identify a device.

2.4.2 Presenting a homogeneous fingerprint

Another defence is to make all devices on the web present the same fingerprint. This is exactly the approach chosen by the Tor Browser [181] also known as TBB (the Tor Browser Bundle) which uses the Tor network.

The theory

At its heart, Tor is a distributed network of relays that relies on the concept of *onion routing* which was introduced by Reed et al. [158]. It allows the connection between a client and a server to remain anonymous by routing the network packets of an individual through a series of different nodes. Dingledine et al. gave birth to the Tor network as it is known today by expanding on the original design of onion routing with concepts like forward secrecy or integrity checking [66].

While the Tor network prevents an attacker from finding out the real IP address of a client, it does not modify the actual content of an HTTP request. If a cookie ID or a browser fingerprint is present in the payload, a server can uncover the true identity of a user. To fix this problem, the Tor Browser was developed. As detailed by the official design document [184], the Tor Browser follows two main categories of requirements. The *Security requirements* are designed to make sure that the browser is properly configured to use the Tor network. The *Privacy requirements* are concerned with reducing the linkability between two websites These requirements notably include a *Cross-Origin Fingerprinting Unlinkability* section which specifically targets browser fingerprinting. In the design document, they analyse the best strategy to fight against fingerprinting between randomization and uniformity. While they acknowledge that randomization can be effective with PriVaricator [123] cited as an example, they list several strong reasons why uniformity or the *one fingerprint for all* strategy is better suited for the Tor Browser:

- **Evaluation and measurement difficulties** Introducing randomization may appear interesting on paper but it is very difficult to objectively measure its effectiveness. They add that it can also add a false sense of security because fingerprinters could ignore randomized information or try to create a more stable fingerprint by either removing the randomness, modeling it, or averaging it out. This concern is comparable to the one introduced with cross-browser fingerprinting. By ignoring browser-specific information, Boda et al. [41] and Cao et al. [48] are still able to identify devices on the Internet. In the case of the Tor Browser, if not enough attributes are randomized, users could still be identifiable.

- **Randomizing attributes related to the hardware** While it is easy to replace a browser-populated value like *navigator.platform* or *navigator.userAgent*, it is much harder to properly randomize attributes that are related to the hardware. The browser must introduce enough

randomness so that it behaves convincingly like other devices and the exact hardware properties must be blurred so that they cannot be used for identification.

- **Usability issues** We explained earlier in this chapter that browser fingerprinting is born from the information given by browsers to improve the user experience. If one starts to modify browser attributes extensively, it can completely break browsing for a user as the browser will not receive a web page that is compatible with its supported technologies.
- **Performance costs** Randomization increases performance costs. In the case of browser fingerprinting, modifying a large part of a fingerprint can induce heavy performance costs as the fingerprinting surface is getting larger with time.
- **Increased vulnerability surface** As described by the design document, improper randomization can introduce new fingerprinting vectors that could be exploited by third parties to facilitate the fingerprint process.

With all these reasons, Tor developers opted for the uniformity strategy. They modified Firefox to provide a built-in protection against known fingerprinting vectors. The design document lists 24 different modifications that have been introduced in the Tor Browser. The most notable ones are the blocking of the Canvas and WebGL API, the complete removal of plugins, the inclusion of a default bundle of fonts to prevent font enumeration and the modification of the user-agent along with HTTP headers. Whether a user is on Windows, Mac and Linux, the Tor Browser will always report that the device is on Windows.

The reality

While the Tor Browser can be considered as one of the strongest defences against browser fingerprinting, it still presents some shortcomings. First, the fingerprint exposed by the Tor Browser is known and easily identifiable. At the time of writing, the latest stable version (6.51) presents the following user-agent.

```
Mozilla/5.0 (Windows NT 6.1; rv:45.0) Gecko/20100101 Firefox/45.0
```

Since TBB is based on Firefox ESR (Extended Support Release) [89], it still presents version 45 of Firefox. The current non-ESR version that most users have is currently under version 52. Then, if we look at the screen resolution, the values are rounded by default around 1,000 pixels. TBB users can be identified easily as most users will present common values like 1024x768 or 1920x1080. With only these two values coupled with the IP addresses from known Tor exit nodes, a website can determine with high accuracy if the user is using the Tor browser or a standard one. While this may not be important with respect to identification as one website cannot distinguish one Tor user from another one, it can still impact their browsing experience as shown by Khattak et al. [109]. They reported that 3.67% of the top 1,000 Alexa sites either block or offer degraded service to Tor users to reduce Internet abuse.

The second problem with Tor browser fingerprints is their brittleness. As it is not possible to restrict every Tor users to the exact same fingerprint, differences can be detected between browsers. The most notable one is the screen resolution. When first launched, the Tor Browser window has a size of 1,000x1,000. However, if the user decides to maximize the window, the browser displays the following message: “Maximizing Tor Browser can allow websites to determine your monitor size, which can be used to track you. We recommend that you leave Tor Browser windows in their original default size.”. If the user has an unusual screen resolution, this information could be used to identify him as he will be the only Tor user with this screen resolution.

The third and final problem is that there are still detectable differences between operating systems running the Tor Browser. The design document notes that they intend to reduce or eliminate OS type fingerprinting to the best extent possible but they add that the efforts in that area is not a priority. While this may provide very few information compared to other fingerprinting vectors, OS differences are yet an additional vector that can be used to distinguish a user from the pool of all Tor users.

In the end, developers of the Tor Browser have made some very strong modifications to limit as much as possible the fingerprintability of the browser. If users stick with the default browser

fingerprint that most users share, it provides the strongest protection against known fingerprinting techniques. However, if one starts to deviate from this one and unique fingerprint, the user may end up being more visible and more easily trackable than with a standard browser like Chrome or Firefox.

2.4.3 Decreasing the surface of browser APIs

The last defence detailed in this chapter is to decrease the surface of browser APIs and reduce the quantity of information that can be collected by a tracking script. One approach is to simply disable plugins so that additional fingerprinting vectors like Flash or Silverlight are not available to leak extra device information.

Another straight-forward way is to simply not run tracking scripts. One can go into the browser preferences and disable the execution of JavaScript code for all web pages. However, by doing so, the user will meet a static and broken web where it is impossible to login to most services. An alternative is to use a browser extension like NoScript which uses a whitelist-based blocking approach that prevents exploitation of security vulnerabilities [126]. By default, all JavaScript scripts are blocked and it is up to the user to choose which scripts can run. The major problem with NoScript is that it is hard sometimes to distinguish which scripts are necessary to display a web page correctly and which domains belong to unwanted third parties. In the end, the user ends up authorizing all scripts on the page and, in doing so, allow without knowing the execution of fingerprinting scripts.

Another approach is to use ad and tracker blockers which block scripts and domains based on curated lists. When a page is loaded, the extension analyses its content. If it finds a script or a domain that is present in one of its lists, it will block it. The most popular addons based on this workflow are Adblock Plus [16], Ghostery [96], uBlock Origin [192] and Disconnect [67]. They all have their own methods of updating their lists. AdBlock Plus and uBlock Origin mostly rely on lists updated by the community like EasyList [71] and Ghostery has its own crawling system to detect new trackers. One of the main downside of this type of extensions is that it can take time a lot of time before a new script is detected and blocked, leaving the user vulnerable in the meantime. Moreover, the lists require a significant amount of maintenance as they must be constantly updated and tweaked depending on new actors and user feedback. This approach is also not safe from false positives as some benign scripts can be misclassified leading to web pages being completely broken. For these reasons, developers are turning to new techniques to detect and classify scripts found online.

Yu et al. proposed a concept in which users collectively identify unsafe data elements and report them to a central server [208]. In their model, all data elements are considered unsafe by default when they are first reported. Then, if enough users report the same value for a given script, the data elements are considered to be safe as it cannot be used to uniquely identify a user or a group of users. To assess the effectiveness of their approach, they implemented their technique inside the Cliqz browser [54] where they collected anonymous data from 200,000 users. For one week, all users formed a global intelligence network where each browser contributed in the classification of scripts. By comparing their approach with results from the Disconnect extension, they concluded that some popular blocking extensions are overly aggressive as they block requests that contain no unsafe data. They were also able to detect new scripts that were sending unsafe data but they were not blocked by the Disconnect extension. With these results and the recent addition of Ghostery [55], the future of Cliqz is promising and one to look out for.

The EFF who were behind the original Panopticlick study [72] released in 2014 an extension called Privacy Badger [144]. The tool is similar to the approach chosen by Yu et al. to identify unsafe scripts but instead of relying on a global network of users, everything is computed locally by the extension. The list of blocked scripts is somehow unique to each instance of Privacy Badger as it is being built alongside the websites that the user visits. However, the main downside of Privacy Badger is that the heuristic creation of blocking rules is too aggressive and leads to a high number of unresponsive websites as reported by [113].

In terms of blocking protection, the last approach consists in disabling browser functions and even entire APIs to prevent trackers from using them. This way, tracking scripts cannot collect values that could help them differentiate one device from another. For example, an extension like

CanvasBlocker [46] on Firefox can disable the use of the Canvas API for all websites. A script is then unable to collect any values if it interacts with this API. The extension offers other modes where websites are still allowed to render images in the browser (i.e. writing is allowed) but all the reading functions like *toDataURL* are blocked so that scripts cannot get access to the image content. Another example is the Brave browser [42] which provides a built-in fingerprinting protection [43]. If it is activated, the following APIs are blocked: Canvas, WebGL, AudioContext, WebRTC against IP leakage and Battery Status. Finally, the Tor Browser [181] which is explained in greater details in Section 2.4.2 also blocks by default APIs like Canvas or WebGL to prevent scripts from learning additional information.

2.4.4 Summary of existing defences

Table 2.1 provides a summary of all the defences detailed in this chapter. While some solutions provide very strong protection against browser fingerprinting, it is often at the cost of usability as we can see for example with NoScript or the Tor Browser. If we take a look at the scientific publications, we can see that the biggest challenge met by researchers is to provide a complete coverage of modified attributes as the slightest mismatch render users more visible to trackers. A solution can be rendered useless in a matter of months as browsers are constantly updated and new APIs are surfacing frequently.

Our contributions: Blink and FPRandom

In Chapter 4, we detail our contributions to the domain of defences against fingerprinting. The first one is called Blink [1] and works at the operating system layer. From a pool of thousands of components composed of plugins, fonts, browsers and operating systems, Blink assembles these components at runtime to obtain a new fingerprint for each browsing session. Since Blink relies on real components that truly run on the device, it does not present the incomplete coverage problem met by many solutions as the exhibited fingerprints are genuine with no inconsistencies. The second solution is FPRandom [3] and works at the browser level. By modifying the source code of Firefox, we are able to introduce noise into very specific fingerprinting routines. The novelty of the solution is that we exploit the flexibility of the JavaScript standard to target advanced fingerprinting techniques.

2.5 Conclusion

The web is built on the principle that it is device agnostic. Whatever device is used to connect to the Internet, anyone is able to browse its content and communicate with the rest of the world. To make it a reality, the web relies on universal languages and protocols like HTML, CSS, JavaScript or HTTP to support an incredibly rich ecosystem of devices from tablets to phones to laptops. However, as devices share parts of their characteristics and configuration to optimize the user experience, this diversity opened the door to browser fingerprinting. By collecting information on different layers of a system, one is able to build a signature of a device and identify it among many other ones. The implications on privacy are then really strong as the fingerprinting process is completely transparent and the user has no control over it.

We showed in this chapter that the information contained in a fingerprint can come from many sources: HTTP headers, browsers APIs or plugins if they are installed. While some attributes do not give enough details to differentiate one device from another, others are much more revealing by providing the exact version of a kernel or the model of a hardware component. In Chapter 3, we provide a much needed update on the state of device diversity on the web with the recent booming of mobile devices and the introduction of new browser APIs.

By going through a varied list of defences against fingerprinting, we showed that there is no simple and ultimate solution to mitigate tracking. Finding a good solution is always a compromise between protection and usability and adjusting correctly these two dimensions is never easy as we saw in the advantages and the weaknesses of spoofers on one side and the Tor Browser on the other. While some focus on blocking scripts, others try to modify the content of fingerprints. On paper,

	Solution	Citation	Type	Comments
Scientific publications	FP-Block	[185]	M	+: Separation of web identities −: Incomplete coverage
	FPGuard	[83]	M	+: Detection and prevention of fingerprinting −: Lack of details
	Fiore et al.	[88]	M	+: Aims at creating consistent fingerprints −: Incomplete coverage
	DCB	[37]	M	+: N:1/1:N strategies, changes at each session, creation of groups with similar configurations −: Incomplete coverage (?)
	PriVaricator	[123]	M	+: Custom randomization policies −: Incomplete coverage
	Changing browsers	[41, 48]	M*	+: Presents distinct and genuine fingerprints −: Can be bypassed
	Cliqz browser	[54, 208]	BS	+: Strong protection against scripts with unique identifiers −: Relies on a central server
Online tools	Canvas Defender	[47]	M	+: Modifications consistent across a browsing session −: Only canvas
	Random Agent Spoofer	[157]	M	+: Uses real database of browser profiles −: Incomplete coverage
	Tor Browser	[181]	U,BA	+: Very strong protection against fingerprinting −: Tor fingerprint is brittle
	NoScript	[126]	BS	+: Blocks all JavaScript scripts −: Blocks all JavaScript scripts
	Adblock Plus	[16]	BS	+: Extensive blocking list −: Relies on lists of known trackers
	Ghostery	[96]	BS	+: Very extensive blocking list −: Relies on lists of known trackers
	uBlock Origin	[192]	BS	+: Extensive blocking list −: Relies on lists of known trackers
	Disconnect	[67]	BS	+: Very extensive blocking list −: Relies on lists of known trackers
	Privacy Badger	[144]	BS	+: Heuristics-based approach −: Blocking may be too aggressive
	Canvas Blocker	[46]	BA	+: Blocks the entire canvas API −: Other vectors can still be used
	Brave browser	[42]	BA	+: Blocks the entire Canvas, WebGL, AudioContext and WebRTC APIs −: Other vectors can still be used
Contributions	Blink	[1]	M	+: Produces genuine and diverse fingerprints with no inconsistencies −: Take HDD space
	FPRandom	[3]	M	+: Introduces noise into the Canvas, AudioContext APIs and randomises the enumeration order of JavaScript objects −: Other vectors can still be used

Table 2.1: Summary of existing defence solutions. M = Modifying the fingerprint content. M* = Modifying the fingerprint content by switching browsers. U = Universal fingerprint. BS = Blocking Scripts. BA = Blocking APIs.

all defence solutions are sound but the biggest difficulties lie in the implementation as researchers and developers are faced with the constant updates and evolution of the web as a platform. In Chapter 4, we provide our own contribution to the domain. With Blink, we address the problem of incomplete coverage as we detail a solution that synthesizes genuine fingerprints at runtime thanks to a large pool of browsers, plugins, fonts and operating systems. With FPRandom, we explore a new defence approach by exploiting the flexibility of modern web browsers to mitigate advanced fingerprinting techniques.

Finally, it remains to be seen how browser fingerprinting will be used in the future as the literature has mostly focused in recent years on tracking and mitigating the collection of data. Positive use of this technology for better security is definitely an avenue where browser fingerprinting will grow as many companies are already adding fingerprinting-related techniques to complement their security solutions. In Chapter 5, we bring our own solution to reinforce a multi-factor authentication scheme. Through a challenge-response mechanism, we use the Canvas API of modern browsers to assert the identity of a device at login time. The use of canvas fingerprinting enables us contrary to other techniques to fight off common attacks as we can generate a unique challenge for each connection of a client.

Chapter 3

Investigating browser fingerprinting

Since the Panopticlick study shed light on browser fingerprinting in 2010, web browsers have undergone important changes. The development of HTML5 pushed the limits of what is possible online with the introduction of new APIs like Canvas, WebGL or Web Audio. Mobile browsing has witnessed an incredible growth in the past few years as it overtook desktop browsing for the first time in 2016 [114]. In this changing online landscape, we launched the *AmIUnique.org* website in November 2014 to collect genuine browser fingerprints from online visitors. Our main goal is to investigate the current state of browser fingerprinting and report on the impact of recent technological trends. Especially, we want to study the following areas:

- Verify the results of previous studies on the diversity of devices on the web.
- Assess the impact of new attributes found in the literature.
- Analyse the impact of browser fingerprinting on mobile devices.

This chapter is organised as follows. In Section 3.1, we introduce how we collect browser fingerprints and explain the challenges encountered to support older or legacy browsers. Then, we detail in Section 3.2 what we collect along with detailed statistics for each attribute. In Section 3.3, we assess the impact of the main techniques that emerged since 2010 before investigating the uniqueness of mobile device fingerprints in Section 3.4. We simulate in Section 3.5 possible technological evolutions and we analyse the linkability of browser fingerprints through time in Section 3.6. Finally, we report on the impact of the AmIUnique website on both users and developers in Section 3.7 before concluding this chapter in Section 3.8.

3.1 Building a website for all browsers

3.1.1 Collecting fingerprints

AmIUnique.org is a website dedicated to browser fingerprinting, aimed both at collecting data about device diversity and at informing users about the privacy implications of fingerprinting. All visitors are informed of our goal with links to both our privacy policy and FAQ sections, and they have to explicitly click on a button to trigger the collection of their device's fingerprint. From the very beginning, we released the code of the website as open source to ensure full transparency of the fingerprinting procedure[1].

The workflow to collect browser fingerprints on the AmIUnique website is simple. When visitors land on the homepage, they are presented with the "View my browser fingerprint" button as seen in Figure 3.1. When they click on it, they are redirected to the "My fingerprint" page where the collection process takes place. The amount of information that is sent to the server then depends on the browser and its configuration.

1. If users have disabled the use of JavaScript, we will only collect HTTP headers.

[1] https://github.com/DIVERSIFY-project/amiunique

2. If JavaScript is enabled, we will collect HTTP headers + JavaScript attributes.
3. If JavaScript is enabled and if Flash is present and enabled, we will collected HTTP headers + JavaScript attributes + Flash attributes.

When our fingerprinting script finishes the collection operation, it sends the fingerprint to the server so that it is stored in our database. The server then sends back the percentage of fingerprints sharing the exact same value for each attribute. This lets users know if some attributes are unique so that they can improve their online privacy by tweaking their browser configuration.

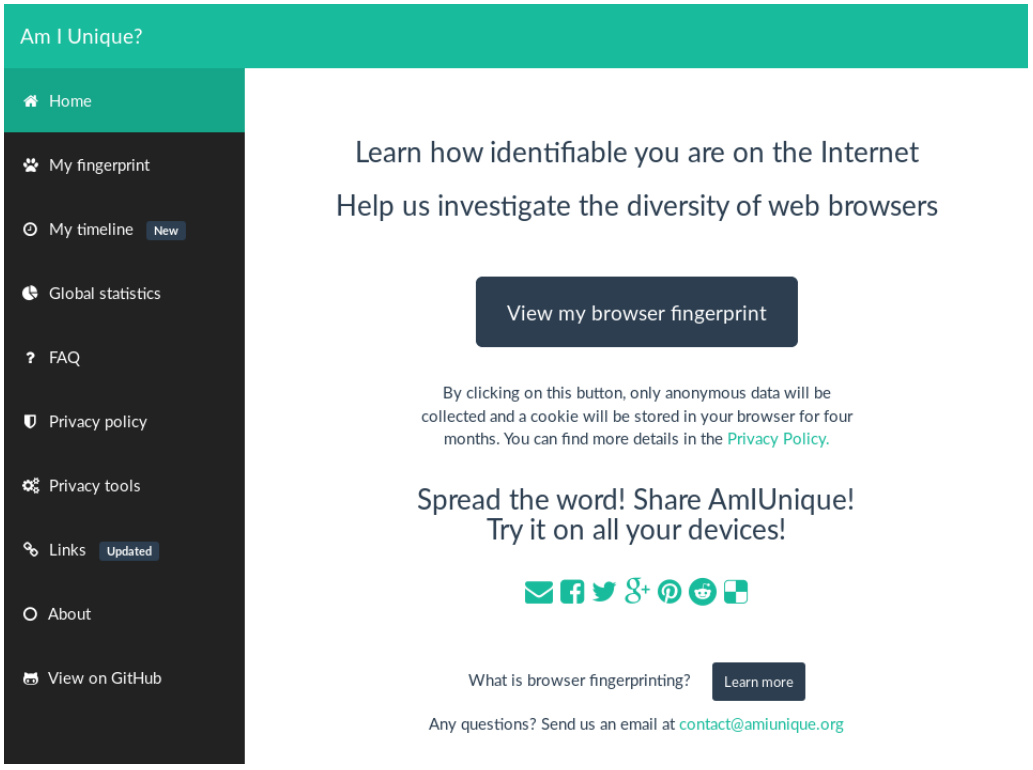


Figure 3.1: Landing page of the AmIUnique.org website

To prevent collecting multiple copies of the same fingerprint from the same user, we store a cookie on the user's device with a unique ID, and we also keep a hashed version of the IP address. These two pieces of information allow us to identify returning devices, which represent a negligible part of our dataset.

3.1.2 Supporting legacy technologies

In order to collect meaningful data on the diversity of devices on the web, it is imperative that the site behaves correctly on the widest range of browsers possible. The most challenging part of building AmIUnique was to support older browsers as they can react very differently to the exact same JavaScript code. For example, it is estimated that 5 to 7% of desktop PCs are still running Windows XP in April 2017 according to Net Applications [64] and StatCounter [65]. When it was released, Windows XP was accompanied with Internet Explorer 6 and updates brought Internet Explorer 8 to the OS in 2009. When designing a fingerprinting script with these legacy browsers in mind, one has to test them to be sure that they work as intended. The JavaScript code may throw unexpected errors as modern keywords or data structures were not supported at the time. Some attributes may also be collected differently like with plugins in Internet Explorer as ActiveX controls are used instead of the NPAPI architecture. The same reasoning has to be applied when configuring the server for HTTPS connections. Older browsers like Internet Explorer do not support modern ciphersuites so the security of the HTTPS connection had to be downgraded so that they could still connect to AmIUnique through less secure ones.

All in all, the core of the AmIUnique website was developed in the span of two weeks but it required an additional month to extensively test and adapt it to less modern browsers as the landscape of the web 10 years ago was much different than how it is today.

3.2 Statistics on 118,934 fingerprints

AmIUnique was launched in November 2014. Thanks to websites like Slashdot [23], Clubic [20], Framasoft [21] and social media channels like Twitter [22], the website quickly gained some traction and it attracted several tens of thousands of visitors in the first few months. The study presented in this chapter is based on fingerprints collected between November 2014 and February 2016. Given that our work focuses on fingerprinting modern browsers and at analysing the importance of the attributes in Table 3.1, we do not consider fingerprints with no JavaScript. As of February 15th, 2016, we collected 142,023 fingerprints, which were then reduced to 118,934 once we removed the fingerprints without JavaScript for this study. At the time of writing in June 2017, more than 390,000 browser fingerprints have been collected. We provide in Appendix B.5 an update of our 2016 study with descriptive statistics from this larger dataset.

The first part of this section presents the set of attributes that we collect in our browser fingerprinting script. Then, we give a few general descriptive statistics about the fingerprints that serve as our dataset. We finish this section with a series of tests to compare our dataset with the only other available set of fingerprint statistics, provided by Eckersley in 2010 [72].

3.2.1 Fingerprinting script

We implemented a browser fingerprinting script that exploits state-of-the-art techniques [13, 116] as well as some new browser APIs. The complete list of attributes is given in the 'Attribute' column of Table 3.1. The 'Source' column indicates the origin of each attribute (HTTP, JavaScript or Flash). The 'Distinct values' and 'Unique values' columns give a global overview of the most discriminating attributes in a fingerprint. Finally, the last column displays a complete example of a browser fingerprint. The top 10 attributes have been presented by Eckersley. Most of the 7 attributes at the bottom of the table have been discussed in other works. Yet, we are the first to collect them on a large scale basis and to combine them as part of a fingerprint. We detail these 7 attributes below:

- List of HTTP headers: When connecting to a server, browsers send the user-agent, the desired language for a web page, the type of encoding supported by the browser, among other headers. Some software and browser extensions modify or add headers, giving extra details about the device's configuration. Being defined in the HTTP protocol, these headers can always be acquired by the server and do not depend on JavaScript.
- Platform: The value in the "*navigator.platform*" property provides information about the user's operating system. While this information is already in the user-agent, we collect the 'platform' value to detect modified or inconsistent fingerprints, e.g., in case the returned value is different from the one in the user-agent.
- Do Not Track/Use of an ad blocker: These two attributes are directly related to privacy and the values can help us differentiate privacy-conscious users from others.
- WebGL Vendor and Renderer: Described by Mowery et al. [116], these two attributes were added with the HTML WebGL API to give information on the underlying GPU of the device. We provide extensive details about the contents of these attributes in Section 3.3.
- Canvas: Introduced by Acar et al. [13] and fully explained in Section 3.3.1, the HTML5 Canvas element gives us the ability to perform tests on both the hardware and the operating system by asking the browser to render a picture following a fixed set of instructions.

It should be noted that the WebGL Vendor and WebGL Renderer attributes were added after our site was launched. We isolated the results obtained from these two attributes (values collected after fingerprint number 45,474).

Table 3.1: Browser measurements of AmIUnique fingerprints with an example

Attribute	**Source**	**Distinct values**	**Unique values**	**Example**
User agent	HTTP header	11,237	6,559	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/ 41.0.2272.118 Safari/537.36
Accept	HTTP header	131	62	text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Content encoding	HTTP header	42	11	gzip, deflate, sdch
Content language	HTTP header	4,694	2,887	en-us,en;q=0.5
List of plugins	JavaScript	47,057	39,797	Plugin 1: Chrome PDF Viewer. Plugin 2: Chrome Remote Desktop Viewer. Plugin 3: Native Client. Plugin 4: Shockwave Flash...
Cookies enabled	JavaScript	2	0	yes
Use of local/session storage	JavaScript	2	0	yes
Timezone	JavaScript	55	6	-60 (UTC+1)
Screen resolution and color depth	JavaScript	2,689	1,666	1920x1200x24
List of fonts	Flash plugin	36,202	31,007	Abyssinica SIL,Aharoni CLM,AR PL UMing CN,AR PL UMing HK,AR PL UMing TW...
List of HTTP headers	HTTP headers	1,182	525	Referer X-Forwarded-For Connection Accept Cookie Accept-Language Accept-Encoding User-Agent Host
Platform	JavaScript	187	99	Linux x86_64
Do Not Track	JavaScript	7	0	yes
Canvas	JavaScript	8,375	5,533	
WebGL Vendor	JavaScript	26	2	NVIDIA Corporation
WebGL Renderer	JavaScript	1,732	649	GeForce GTX 650 Ti/PCIe/SSE2
Use of an ad blocker	JavaScript	2	0	no

We tested other attributes for inclusion in the fingerprints, but the results were inconclusive and we decided to discard them. We designed a test that renders 3D volumes through the WebGL API, as first tested by Mowery et al. [116]. However, after an early analysis of more than 40,000 fingerprints, the test proved to be too brittle and unreliable since a simple page reload with a different window size on a single device could change the value of this test. Appendix B.2 goes into more details on this WebGL test. We also tested the collection of information based on the device's hardware performance, like the Octane JavaScript benchmark, but they proved to be too long and too intensive to execute. Finally, we included other Flash attributes that proved to be useful to detect inconsistencies, but did not increase fingerprint uniqueness. More details can be found in Appendix B.3.

3.2.2 Descriptive statistics

Tables 3.1 and 3.2 summarize the essential descriptive statistics of the AmIUnique dataset. It should be noted that because our website focuses on a very specific subject, our visitors are likely saavy Internet users who are aware of potential online privacy issues. Hence, our data is biased towards users who care about privacy and their digital footprint, and their devices might have fingerprints different than those we could collect from a more general audience. Table 3.2 presents the distribution of plugins, fonts and headers in our dataset. To obtain these numbers, we decomposed each list of values into single elements and we studied how common they are by looking at the number of fingerprints in which each element is present. We divided the results from the plugins, fonts and headers into three categories: the ones that belong to less than 1% of collected fingerprints, the ones present in less than 0,1% of fingerprints, and the ones that appear in only one or two fingerprints.

Unique and distinct values

The 'Distinct values' column in Table 3.1 provides the number of different values that we observed for each attribute, while the 'Unique values' column provides the number of values that occurred a single time in our dataset. For example, attributes like the use of cookies or session storage have no unique values since they are limited to "yes" and "no". Other attributes can virtually take an infinite number of values. For example, we observed 6,559 unique values for the user-agent attribute. This is due to the many possible combinations between the browser, its version and the operating system of the device. It is extremely likely that visitors who use an exotic OS with a custom browser, such as Pale Moon on Arch Linux, will present a very rare user-agent, thus increasing the likelihood of being identified with just the user-agent.

These numbers show that some attributes are more discriminating than others, but they all contribute to building a unique and coherent fingerprint.

Plugins

We observed 2,458 distinct plugins, assembled in 47,057 different lists of plugins. They cover an extremely wide range of activities, as for example, reading an uncommon file format in the browser (e.g., FLAC files with the VLC Browser plugin), communicating with an antivirus or a download client, launching a video game directly in the browser, site-specific plugins for added functionality, etc. Some plugins are so specific that they leak information beyond the computer, like the company the user works for or the brand of smartphone, camera or printer he or she uses. 97% of plugins appear in less than 1% of collected fingerprints and 89% in less then 0,1%. A lot of plugins are created for precise and narrow uses allowing their users to be easily identified.

Fonts

We observed 221,804 different fonts, assembled in 36,202 different lists of fonts. This really high number shows the incredible wealth that exists: fonts for support of an additional alphabet, fonts for web designers, fonts for drawing shapes and forms, fonts for different languages, etc. On average, a Windows or Mac user has two to three times the amount of fonts of a Linux user. Also, 97% of fonts appear in less than 0,1% of fingerprints and a little less than 2/3 of them are only in one or two fingerprints. These percentages show how efficient a list of fonts can be for fingerprinting and transitively how critical it can be for users who want to protect their privacy. However, this list is provided through the Flash plugin, which is progressively disappearing from the web. We will see in Section 3.5 that removing access to the list of fonts has a small impact on identification.

HTTP headers

We observed 222 different HTTP headers, assembled in 1,182 different lists of headers. New headers are added to the standardized ones for different reasons and from different sources. Some examples include the following:

- The browser. For example, the Opera browser on smartphones adds a *X-OperaMin-Phone-UA* header, and the Puffin browser adds a *X-Puffin-UA* header.
- A browser extension. For example, the FirePHP extension for Firefox adds the *x-FirePHP* and the *x-FirePHP-Version* headers to each HTTP request.
- The network on which you are connected. Some headers show the use of proxies or protection systems.

As indicated in Table 3.2, 182 headers out of 222 appear in less than 0,1% of the collected fingerprints, and 92 of them come from only one or two fingerprints. These statistics mean that some HTTP headers are highly discriminating and their presence greatly affects the uniqueness of one's fingerprint.

Table 3.2: Summary of statistics

Attribute	Total	<1% FP	<0,1% FP	< 3 FP
Plugin	2,458	2,383 (97%)	2,195 (89%)	950 (39%)
Font	223,498	221,804 (99%)	217,568 (97%)	135,468 (61%)
Header	222	205 (92%)	182 (82%)	92 (41%)

3.2.3 Statistical validity of the dataset

This section presents a series of tests to compare our dataset with the fingerprinting statistics provided by Eckersley in 2010.

Mathematical treatment

Entropy

We use entropy to quantify the level of identifying information in a fingerprint. The higher the entropy is, the more unique and identifiable a fingerprint will be.

Let H be the entropy, X a discrete random variable with possible values $\{x_1, ..., x_n\}$ and $P(X)$ a probability mass function. The entropy follows this formula:

$$H(X) = -\sum_i P(x_i) \log_b P(x_i)$$

We use the entropy of Shannon where $b = 2$ and the result is in bits. One bit of entropy reduces by half the probability of an event occurring.

Normalized Shannon's entropy

To compare both the AmIUnique and Panopticlick datasets, which are of different sizes, we use a normalized version of Shannon's entropy:

$$\frac{H(X)}{H_M}$$

H_M represents the worst case scenario where the entropy is maximum and all values of an attribute are unique ($H_M = \log_2(N)$ with N being the number of fingerprints in our dataset).

The advantage of this measure is that it does not depend on the size of the anonymity set but on the distribution of probabilities. We are quantifying the quality of our dataset with respect to an attribute's uniqueness independently from the number of fingerprints in our database. This way, we can compare the two datasets despite their different sizes.

Table 3.3: Normalized entropy for six attributes collected both by Panopticlick and AmIUnique

Attribute	AmIUnique	Panopticlick
User agent	0.570	0.531
List of plugins	0.578	0.817
List of fonts	0.446	0.738
Screen resolution	0.277	0.256
Timezone	0.201	0.161
Cookies enabled	0.042	0.019

Comparison with Panopticlick

Entropy

Table 3.3 lists the normalized Shannon's entropy for six different attributes for both the AmIUnique and the Panopticlick datasets. For fairness of comparison, we used our dataset in its entirety by keeping fingerprints without JavaScript. We observe that the entropy values for both datasets are similar for all attributes except for the list of plugins and the list of fonts.

For the list of plugins, it is still the most discriminating attribute but a difference of 0.24 is present. It can be explained by the absence of plugins on mobile devices which are increasingly used to browse the web and by the lack of support for the old NPAPI plugin architecture on Chrome since April 2015 (more details in Section 3.5).

For the list of fonts, a noticeable drop of 0.29 occurs because half of the fingerprints in the AmIUnique dataset were collected on browsers that do not have the Flash plugin installed or activated. Since our fingerprinting script collects the list of fonts through the Flash API, this means half of our fingerprints do not contain a list of fonts, reducing its entropy. The absence of Flash can be explained (i) by the lack of Flash on mobile devices; (ii) by the fact that the visitors of AmIUnique are privacy conscious and tend to deactivate Flash. Yet, we notice that the entropy of the list of fonts is still high.

The small value of entropy for the timezone shows that our dataset is biased towards visitors living in the same geographical areas. A higher level of entropy would have meant a more spread distribution of fingerprints across the globe.

Distribution of fingerprints

We compared frequency distributions w.r.t. anonymity set sizes from both datasets and observed very similar trends. We also studied each attribute separately and observed that the most discriminating attributes are still the ones found by Eckersley with the addition of new efficient techniques like canvas fingerprinting. More details on the distributions can be found in Appendix B.4.

3.3 Fingerprinting with the most recent technologies

AmIUnique collects 17 attributes to form a browser fingerprint. Out of the 118,934 fingerprints that we study, 89.4% are unique. In this section, we analyse how the attributes collected with the most recent technologies (7 attributes at the bottom of Table 3.1) contribute to the uniqueness of fingerprints.

3.3.1 Canvas fingerprinting

The canvas element in HTML5 [100] allows for scriptable rendering of 2D shapes and texts. This way any website can draw and animate scenes to offer visitors dynamic and interactive content. As discovered by Mowery et al. [116] and investigated by Acar et al. [13], canvas fingerprinting can be used to differentiate devices with pixel precision by rendering a specific picture following a fixed set of instructions. This technique is gaining popularity in tracking scripts due to the fact that the rendered picture depends on several layers of the system (at least the browser, OS, graphics drivers and hardware).

Our test

The fingerprinting script used by AmIUnique includes a test based on the canvas element. With this image, we collect information about three different attributes of the host device, as discussed below.

Figure 3.2 displays the image that we use, as it is rendered by a Firefox browser running on Fedora 21 with an Intel i7-4600U processor. Our test replicates the test performed by AddThis and described in details by Acar et al. [13]: print a pangram twice with different fonts and colors, the U+1F603 unicode character and rectangle with a specific color. The only adaptation is to change the position of the second string so that it is not intertwined with the first one. More details about this test are discussed below.



Figure 3.2: Example of a rendered picture following the canvas fingerprinting test instructions

Font probing

This test captures OS diversity. The script tells the browser to render the same *pangram* (a string with all the letters of the alphabet) twice. For the first line we force the browser to use one of its fallback fonts by asking for a font with a fake name. Depending on the OS and fonts installed on the device, the fallback font differs. For the second line the browser is asked to use the **Arial** font that is common in many operating systems and is used for the hardware and OS fingerprinting described next.

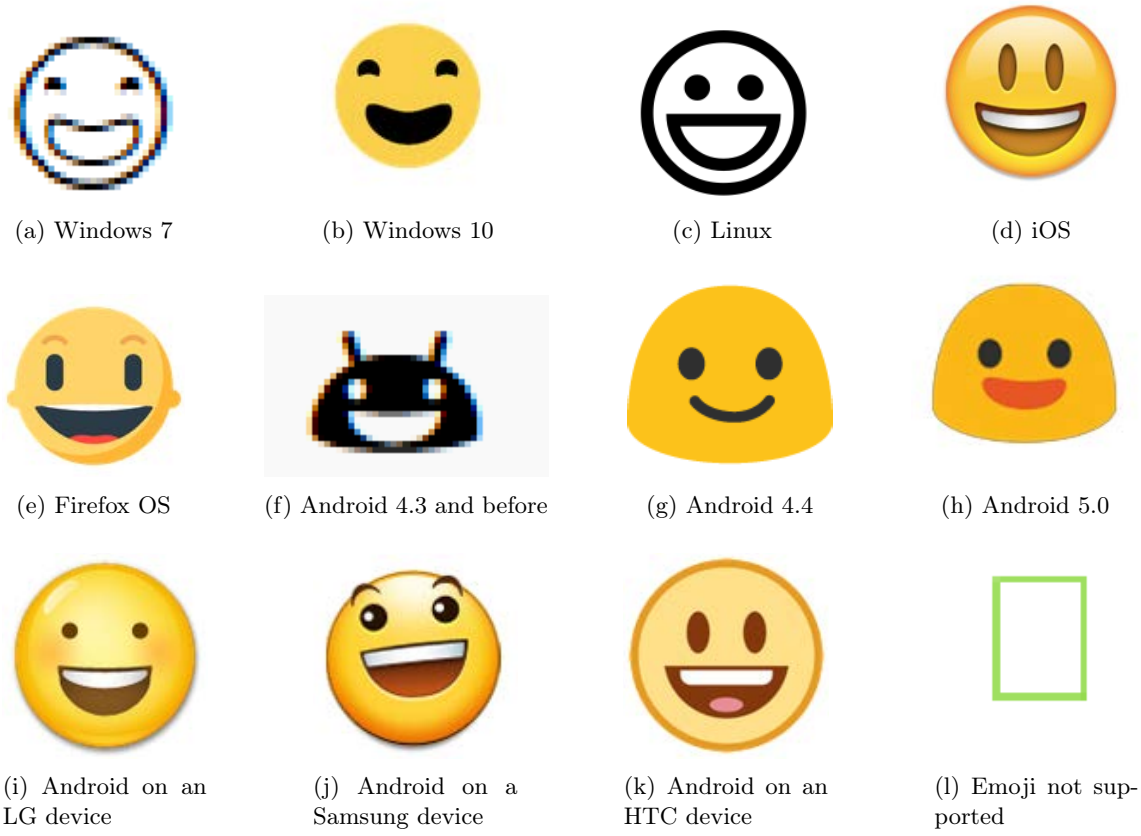
Device and OS fingerprinting

The last character of our string may be the most important one. This character should not be confused with an emoticon, which is a succession of letters, numbers and punctuation marks like ":)" or "<3" to describe an emotion. The character is an **emoji** [75]. Officially introduced in the Unicode standard 6.0 in 2010, emojis are ideograms that represent emotions or activities. The difference with emoticons is that emojis have their own Unicode character and font developers must provide their own implementation for a given emoji w.r.t. its description. Consequently, emojis can be used for fingerprinting because their actual representation differs between systems.

Figure 3.3 shows representations of the "Smiling face with open mouth" emoji on different operating systems and mobile devices. A square means that the browser has not found a single font on the device that supports that emoji. The use of emojis can be a powerful technique to uncover information, especially on mobile devices where phone manufacturers provide their own sets of emojis.

Hardware and OS fingerprinting

As demonstrated by Mowery et al. [116], small pixel-level differences can be detected between browsers when rendering images, even on the same OS and browser. The second line of text of the canvas test uses the Arial font. Although this font has the same dimensions across operating systems, there are visible variations of pixels in the final image due to differences in the rendering process. The process to render an image is complex and depends on both hardware and software (e.g., GPU, rendering engine, graphic drivers, anti-aliasing, OS), and this test is affected by variations in any of these layers. Interestingly, the test is also relatively stable over time because users do not often change the configuration of layers in the rendering process.



(a) Windows 7 (b) Windows 10 (c) Linux (d) iOS

(e) Firefox OS (f) Android 4.3 and before (g) Android 4.4 (h) Android 5.0

(i) Android on an LG device (j) Android on a Samsung device (k) Android on an HTC device (l) Emoji not supported

Figure 3.3: Comparison of the "Smiling face with open mouth" emoji on different devices and operating systems

Influence of canvas fingerprinting for identification

The strength of canvas fingerprinting comes from the fact that it combines the three tests listed before. Alone, as a simple rendered picture, the normalized entropy is at 0.491, putting it in the top 5 of the most discriminating attributes. However, because emojis reveal information about both the OS and the device, it is possible to use canvas fingerprinting to detect inconsistent fingerprints. For example, by checking if the operating system in the user-agent matches the one indicated by the emoji, we can verify inconsistencies in the fingerprint to detect visitors who spoof their fingerprintable attributes. Thus, the added value of canvas fingerprinting is to strengthen the identity of a fingerprint. Moreover, one of the advantages of canvas fingerprinting is that it is stable. You can run it many times on the same computer and you will have the same result every time, with little variance over time (some variations can be observed if the user decides to update drivers for example). In the end, canvas fingerprinting is an important addition to browser fingerprinting. We discuss this technique in greater depths in Chapter 5 where we analyse how it can be used to augment authentication.

3.3.2 WebGL fingerprinting

WebGL [197] uses the Canvas element described before to render interactive 3D objects natively in the browser, without the use of plugins. With the final specifications in 2011, WebGL 1.0 is now supported in all major browsers.

Our test

The WebGL API, through the
WEBGL_debug_renderer_info interface (as the name indicates, it is designed for debugging purposes), gives access to two attributes that take their values directly from the device's underlying graphics driver. AmIUnique's fingerprinting script collects these two properties, namely:

- the WebGL vendor: name of the vendor of the GPU.
- the WebGL renderer: name of the model of the GPU.

These attributes provide very precise information about the device. For example, we collected exact GPU names like "NVIDIA GeForce GTX 660 Ti" or "Intel HD Graphics 3000". These two attributes also indirectly leak information on your OS and its environment. For example, Chrome uses the ANGLE backend [26] on Windows to translate OpenGL API calls to DirectX API calls. Consequently, the following WebGL renderer string indicates that the browser runs on a Windows machine: "ANGLE (NVIDIA GeForce GTX 760 Direct3D11 vs_5_0 ps_5_0)". Same type of leak with the presence of the "OpenGL engine" substring on Mac systems.

Influence of WebGL fingerprinting on identification

The WebGL vendor and renderer had the potential to become a highly discriminating attribute, but two factors greatly hamper its utility. First, not all browsers give the unmasked version of the vendor and renderer. Chrome provides this information by default but Firefox has this information locked behind a browser flag ("webgl.enable-privileged-extensions") and returns a simple "Not supported" with our script. Second, a non-negligible number of devices share the same hardware. For example, a lot of laptops do not have a dedicated GPU and they use the embedded Intel GPU inside their processor. This reduces the uniqueness of some of the values that we can observe. In the end, the WebGL API opens the door to discriminating information but it is not accessible from every browser.

3.3.3 Additional attributes

We collected the following attributes to study their utility to discriminate browsers, to strengthen a fingerprint by verifying values, and to detect inconsistencies.

Platform

Even though the platform attribute does not add new information, it can be used to detect inconsistencies. For example, on an unmodified device, if the browser indicates in its user-agent that it is running on a Linux system, you expect to see "Linux" as the value of the "*platform*" property. Due to the nature of our website that incites users to modify their browser, we flagged 5,426 fingerprints in our dataset as being inconsistent. Some browsers gave completely random values that had no meaning. Others used extensions to mask the platform value. For example, one fingerprint had the value "masking-agent", indicating that the Masking Agent extension for Firefox [111] was installed. Finally, other browsers modified their user-agent to mimic one from another operating system. The problem was that the platform property was not modified and the script was able to identify the true operating system that the user was trying to hide.

Even with its low entropy, the *platform* property can prove useful in cases where it is badly modified because it can make some devices more prone to identification than others with unique or unusual values.

Do Not Track & Ad blocker

These two attributes have a very low-level of entropy, their values are either "Yes", "No" or "Not communicated" (for the DNT preference). Without the Do Not Track attribute, the percentage of unique fingerprints drops by 0.07% which is negligible. The Ad Blocker attribute is slightly better, with a drop of 0.5%, but still insignificant compared to other attributes like the user-agent or the

list of plugins.

To conclude this section, the additional attributes collected by AmIUnique are game changers: they strengthen fingerprints, allow identification through inconsistency detection. They also allow identification even when the list of fonts is inaccessible because of the absence of Flash, and they provide essential information about browsers on mobile devices as it will be detailed in the next section.

3.4 Mobile fingerprint diversity

Given the growth of mobile devices to browse the web, it is essential to analyse how browser fingerprinting behaves in this context. Our analysis of mobile device fingerprinting is based on 13,105 mobile fingerprints. We select these fingerprints from our dataset by analysing the user-agents. If the user-agent contains a substring that is present in a predefined set ('Mobile', 'Android', 'iPhone' or 'iPad'), the fingerprint is selected as a mobile fingerprint, otherwise, it belongs to the desktop/laptop category.

In this section, we first compare desktop/laptop fingerprints with mobile ones. Then, we perform a detailed analysis of mobile fingerprints, looking at differences between browsers and between mobile operating systems.

3.4.1 Mobile and Desktop fingerprint comparison

Using the attributes from Table 3.1, we succeeded in uniquely identifying 90% of desktop fingerprints. This number is lower for mobile fingerprints at 81%, yet still quite effective. At first sight, the overall results are close. However, as we discuss in this section, the discriminating attributes for mobile fingerprints are very different from those for desktop fingerprints. One factor is the lack of plugins in general, and Flash in particular, for mobile devices. We also discuss the importance of the new attributes collected through the HTML5 canvas and WebGL elements on mobile device fingerprinting.

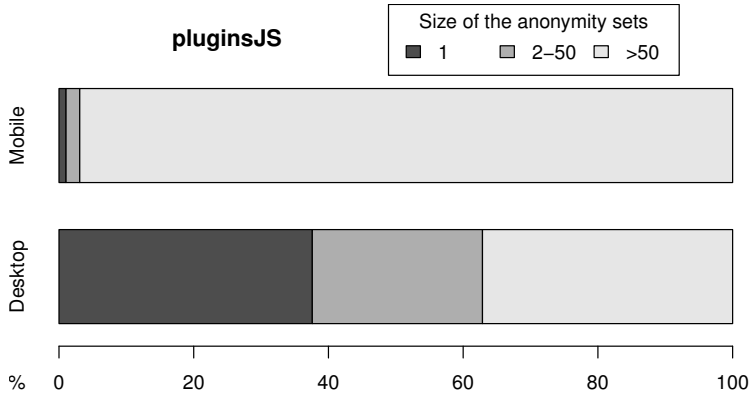


Figure 3.4: Comparison of anonymity set sizes on the list of plugins between desktop and mobile devices

If we take a look at Figure 3.4, we can clearly notice an important difference. For desktops, more than 37% of the collected fingerprints have a unique list of plugins, while it is at 1% for mobile devices. This is due to the fact that mobiles were designed to take full advantage of HTML5 functionalities and do not rely on plugins. For example, Adobe removed the Flash player from the Google Play store in August 2012 as part of a change of focus for the company [90]. Plugins are considered to be unsuitable for the modern web and Google states in their move to deprecate NPAPI support for their Chrome browser that these plugins are a source of "hangs, crashes, security incidents, and code complexity" [169]. This choice helps mobile device users gain some privacy with regards to fingerprint uniqueness. The level of entropy of the plugin attribute is

close to zero (some iOS systems have the QuickTime plugin and some Android systems reported having Flash, possibly from legacy installations). The lack of plugins also reduces information leaks that could come from them. In particular, mobile phones and tablets do not have the Flash plugin, thus all the fingerprint attributes leaked through the Flash API are unavailable.

Despite the unavailability of the two most discriminating attributes from desktop fingerprints (list of fonts and plugins), mobile fingerprints are still very much recognizable. This is due to two main factors: **very rich and revealing user agents and very discriminating emojis**.

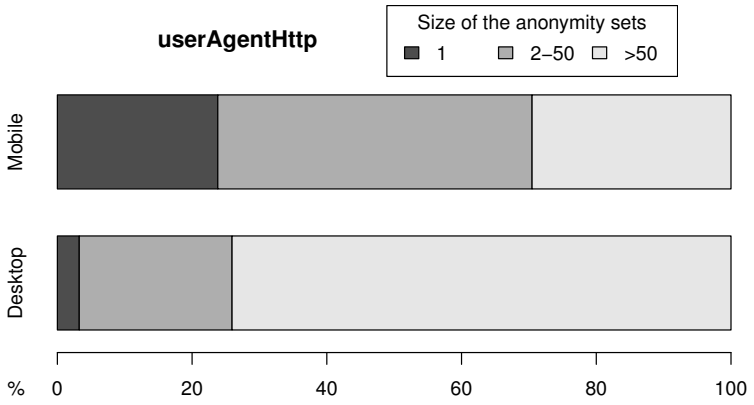


Figure 3.5: Comparison of anonymity set sizes on the user-agent between desktop and mobile devices

Figure 3.5 shows that user-agents found on mobiles are five times more unique than the ones found on desktops. In our dataset, about 1 smartphone out of 4 is instantaneously recognizable with just the user-agent. This is due to two factors:

- Phone manufacturers include the model of their phone and even the version of the Android firmware directly in the user-agent.
  Example:

```
Mozilla/5.0 (Linux; Android 5.0.1; Nexus 5 Build/LRX22C) AppleWebKit/537.36 (
    KHTML, like Gecko) Chrome/40.0.2214.109 Mobile Safari/537.36
```

- On a smartphone, applications are slowly replacing the default browser and they have access to a wide range of personal information after the user has explicitly granted specific permissions. The problem is that any of this information can be exposed by the application for the world to see. We noticed in our dataset that a lot of user-agents collected on mobile devices were sent by an application and not by the native browser.
  Example with the Facebook app where the phone carrier (Vodafone UK) and the exact model of the phone ("iPhone7" = iPhone 6 Plus) is included in the user-agent:

```
Mozilla/5.0 (iPhone; CPU iPhone OS 8_1_1 like Mac OS X) AppleWebKit/600.1.4 (
    KHTML, like Gecko) Mobile/12B436 [FBAN/FBIOS;FBAV/20.1.0.15.10;FBBV
    /5758778;FBDV/iPhone7,2;FBMD/iPhone;FBSN/iPhone OS;FBSV/8.1.1;FBSS/2; FBCR/
    vodafoneUK;FBID/phone;FBLC/en_GB;FBOP/5]
```

  Sometimes, even the model of the phone can give away your phone carrier. One fingerprint reported "SM-G900P". It is a Samsung Galaxy S5 and the "P" is unique to the Sprint phone carrier.

The second highest source of entropy for mobile devices comes from canvas fingerprinting. Mobiles have unique hardware impacting the final rendered picture as explained in Section 3.3.1 and emojis can also be really discriminating between two devices. As seen in Figure 3.3, some manufacturers have their own set of emojis and even between different versions of Android, the emojis have evolved, splitting the Android user base into recognizable groups.

In the end, desktop and mobile fingerprints are somehow equally unique in the eyes of browser fingerprinting even though the discriminating information does not come from the same attributes.

The complete details of attributes' entropy between desktop and mobile devices can be found in Appendix B.1.

3.4.2 Comparison Mobile OS and browsers

More than 97% of mobile fingerprints collected on AmIUnique are either running Android or iOS: 7,416 run on Android and 5,335 on iOS. How diverse is the set of fingerprints coming from both of these operating systems?

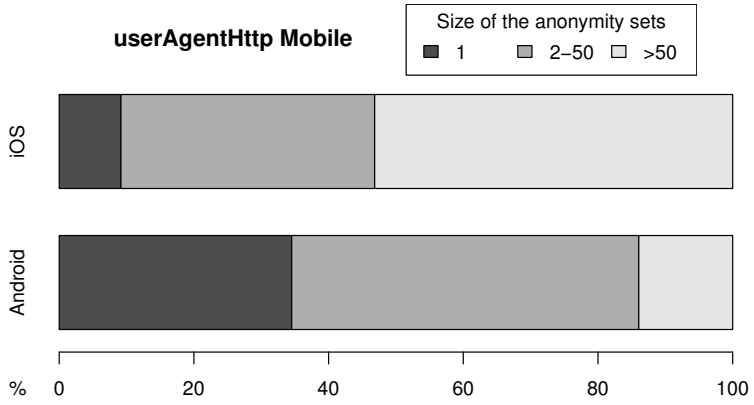


Figure 3.6: Comparison of anonymity set sizes on the user-agent between Android and iOS devices

Figure 3.6 shows the size of anonymity sets for user-agents on both Android and iOS devices. We can see that user agents on Android devices expose more diversity with three times as many users being in an anonymity set of size 1 (9% for iOS devices and 35% for Android devices). This is due to the wealth of Android models available on the market. Moreover, our dataset may not be representative enough of the global diversity of Android devices so these percentages may be even higher in reality. For iOS devices, the diversity is still high but much less pronounced since users share devices with identical configurations. We can notice a trend where half of the collected iOS fingerprints are in really large anonymity sets. The fact that Apple is the only manufacturer of iOS devices shows in this graph.

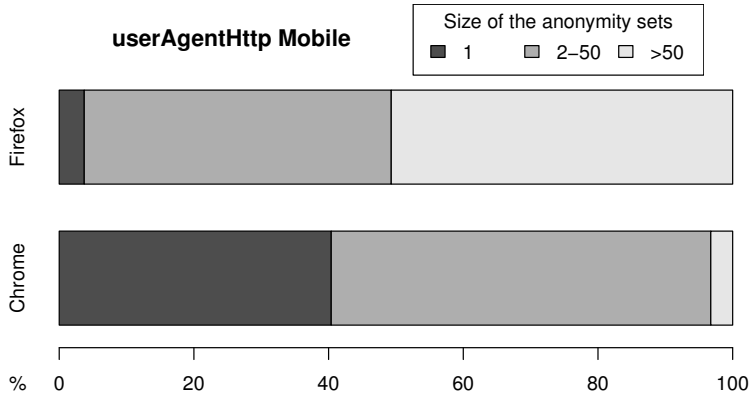


Figure 3.7: Comparison of anonymity set sizes on the user-agent between Chrome and Firefox on mobile devices

We saw in the previous section that user-agents can give really discriminating information on the user's device. Some smarpthones running Android give the exact model and firmware version of their phone. Looking at Figure 3.7, user agents from the Chrome mobile browser are ten times

more unique than user agents from the Firefox browser (40% against less than 4%). This can be explained by the fact that the Chrome browser is the default browser on Android and it is automatically installed on every devices. When a phone manufacturer builds its tailored firmware to be delivered to its clients, the embedded Chrome browser has a user-agent with information on the corresponding phone model and Android version. On the other side, Firefox which can be downloaded from the Google Play Store does not contain this type of information because the store only offers a generic version for every Android mobile and it does not change its user-agent during its installation. Firefox indirectly provides a much better protection against fingerprint tracking by not disclosing device-related information.

You can find below two fingerprints collected from the same device but with a different browser: the first with Chrome, the second with Firefox.

```
Mozilla/5.0 (Linux; Android 4.4.4; D5803 Build/23.0.1.A.5.77) AppleWebKit/537.36 (
    KHTML, like Gecko) Chrome/39.0.2171.93 Mobile Safari/537.36
```

```
Mozilla/5.0 (Android; Mobile; rv:34.0) Gecko/34.0 Firefox/34.0
```

3.5 Assessing the impact of possible technical evolutions

Web technologies evolve very fast, and we have seen in previous sections that some recent evolutions limit fingerprint-based identification (e.g., no Flash on mobile devices), while others open the door to increased identification (e.g., WebGL reveals fine grained information about the GPU).

In this section, we explore 6 potential evolutions that web technology providers (browsers and app developers, standardization organizations) could set up. We demonstrate that they would limit the effectiveness of browser fingerprinting by simulating their impact on our dataset. The first two scenarios are based on current trends in web technologies, while the others are more speculative and based on the observations made in previous sections. It should be noted that we do not estimate the impact of scenarios n°4 and 5 since we can hardly predict which attributes would be affected and how. We also treat scenario n°6 separately, due to its extreme nature.

Scenario n°1 - The definitive disappearance of Flash

The Flash plugin is progressively disappearing. It has been deprecated on all smartphones, tablets and mobile devices used to browse the web. On laptop and desktop browsers, Flash's security flaws have progressively created mistrust in its users. Click-to-play is becoming standard on most browsers. In the meantime, the number of web applications that replace Flash with JavaScript and HTML5 is also growing. These phenomena let us plausibly foresee the definitive disappearance of Flash.

Interestingly, Flash is still present in 80% of our Desktop fingerprints. Among these cases, 71.7% have it activated, 26.3% are using click-to-play protections, and 2.0% block Flash, likely by a browser extension.

Impact of scenario n°1

Figure 3.8 shows the impact of the Flash plugin on fingerprint uniqueness. The "No Flash" bar shows statistics over our complete dataset (for the 60,617 fingerprints that have Flash, we simulate its absence by removing the attributes obtained through Flash). The "Flash" bar is computed with the subset of fingerprints that have Flash, since it is not possible to simulate the presence of Flash on fingerprints that don't have it. We uniquely identify 95% of the browsers that have Flash, while this is reduced to 88% for those without Flash. The sizes of the anonymity sets are notably small, with less than 0.6% of the fingerprints in a set of size 50 or greater. These numbers confirm that browser fingerprinting in a Flash-less future is certainly possible, and that the wealth of fingerprintable attributes compensates for the lack of access to Flash specific attributes.

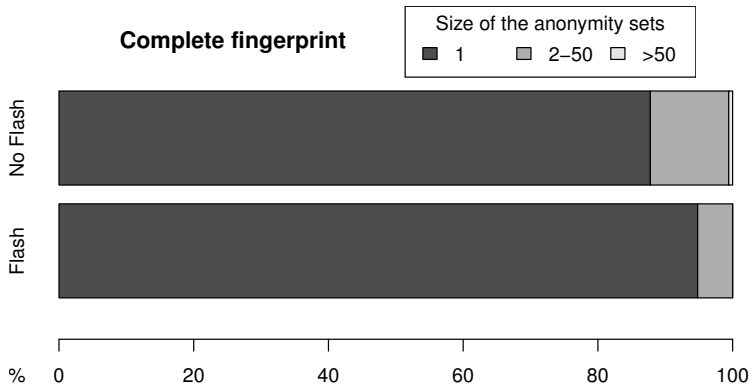


Figure 3.8: Comparison of anonymity set sizes between devices with and without Flash

Scenario n°2 - The end of browser plugins

In 2013, Google decided to stop supporting NPAPI plugins in Chrome and to rely exclusively on the technology embedded in modern browsers and the functionalities offered by HTML5 and JavaScript to let developers extend the browser [169]. This has forced developers to migrate old plugins to newer alternatives [127] or to drop their support. Nevertheless, since its enforcement, it has the advantage of drastically reducing the entropy of the list of plugins. In 2015, version 42 of Chrome deprecated the support of NPAPI plugins by default and version 45 permanently removed their support.

This radical evolution, and the absence of plugins on mobile platforms, lets us foresee a more global evolution where browsers no longer provide a plugin-based architecture. Yet, this is challenging because plugins currently still provide a large number of features (as discussed in Section 3.2, we observed 2,458 different plugins in our dataset). Mozilla had plans to hide unpopular plugins with a whitelist [140] but they did not find a satisfying working solution that would not break websites or functionality. In October 2015, they announced the removal of NPAPI support by the end of 2016 [139] but it was not until the release of Firefox 52 in March 2017 that NPAPI was dropped [119].

Impact of scenario n°2

To estimate the impact of this scenario, we look at the entropy of plugins for Chrome since Google decided to deprecate the support of NPAPI plugins. Figure 3.9 shows the evolution of the normalized entropy of plugins for the stable releases of Chrome since the launch of the AmIUnique website. The last 4 stable versions of Firefox were added for comparison. Up to version 42, the normalized entropy of the list of plugins was above 0.8. Since the release of version 42, the entropy of the list of plugins has dropped below 0.5. This improvement is significant and the effects are getting bigger with the release of version 45 where the NPAPI support is permanently dropped (the entropy is not at zero since there are small differences in the plugin list between operating systems). Removing plugin support definitely impacts desktop fingerprints and it seems that their use in browser fingerprinting is becoming limited.

Scenario n°3 - Adherence to the standard HTTP headers

A major source of information for browser fingerprinting comes from application and system developers that add arbitrary information in headers by either modifying existing headers (e.g., the user-agent) or by adding new ones. Yet, the Internet Engineering Task Force (IETF) has standardized a list of fields for HTTP headers. The current diversity in the contents of the user-agent field results from a very long history of the 'browser wars', but could be standardized today. This scenario explores the possibility that technology providers converge on a standard set of HTTP header fields, and that they follow the standard.

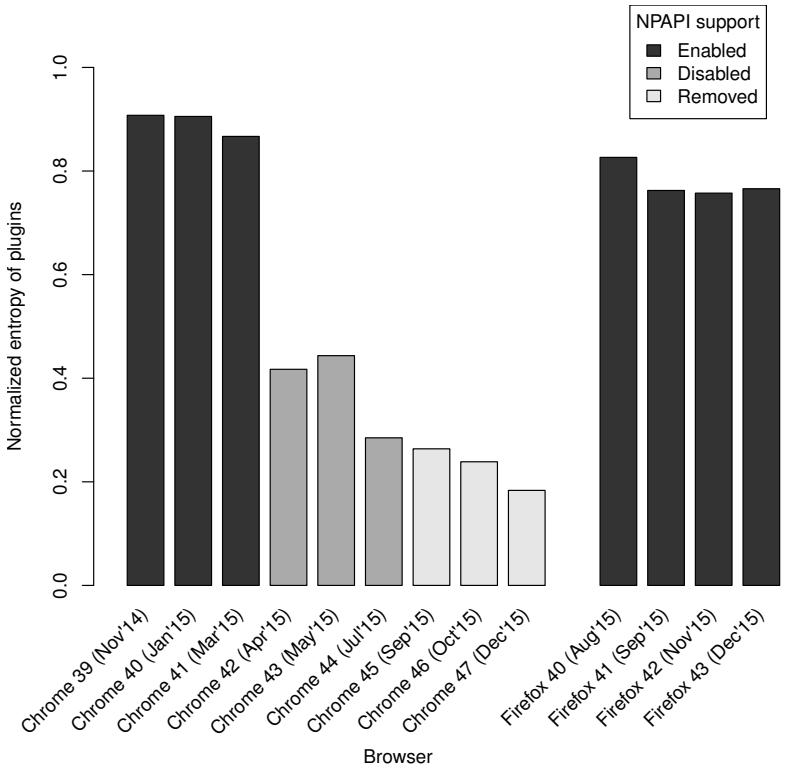


Figure 3.9: Evolution of the normalized entropy of plugins for different browsers on desktop computers

Impact of scenario n°3

To estimate the impact of adherence to standard HTTP headers, we simulate the fact that they are all the same in our dataset. On desktops, the improvement is moderate with a decrease of exactly 8% from 90% to 82% in overall uniqueness. However, on mobile fingerprints, we can observe a drop of 21% from 81% to 60%. This illustrates the importance of headers, and especially the user-agent, for mobile fingerprinting and the fact that generic user-agents are essential for privacy.

Combining scenarios n°1-2-3

The biggest surprise of this analysis comes from combining the 3 scenarios. For mobile devices the results are significant but not overwhelming, the number of unique fingerprints drops by 22%. However for desktop devices, the percentage drops by a staggering 36%, from 90% to 54%. This means that if plugins disappear and if user-agents become generic, only one fingerprint out of two would be uniquely identifiable using our collected attributes, which is a very significant improvement to privacy over the current state of browser fingerprinting.

Scenario n°4 - Reduce the surface of HTML APIs

The potential disappearance of Flash and plugins will occur only if developers find suitable replacements with rich HTML and JavaScript features. Consequently, HTML APIs keep growing, providing access to an increasing number of information about the browser and its environment. As we saw in Section 3.3, the WebGL and canvas elements provide important information for identification. There are potentially many more APIs that leak identifying information.

Setting the best trade-off between rich features and privacy is a critical and difficult choice when setting up new APIs. Developers debate extensively on this kind of trade-off [33]. Yet, it is possible to foresee that future API developments, combined with informed studies about privacy

such as the recent work by Olejnik and colleagues [130], will lead to reduced APIs that still provide rich features.

Scenario n°5 - Increase common default content

This scenario explores the possibility that browser or platform developers increase the amount of default elements, which would be the only ones exposed publicly. For example, we could envision a whitelist of fonts that are authorized to be disclosed by the browser, as suggested by Fifield and Egelman [86]. Such a list would contain the default fonts provided by an operating system. This whitelist of fonts would also include a default encoding for emojis that is common to all versions of the operating system, or even common to all platforms.

This evolution would aim at reducing the amount of information disclosed to external servers. Yet, it should not prevent the users from adding new fonts or new emoji renderings. These customization decisions should be allowed without increasing the risks for privacy.

Scenario n°6 - The end of JavaScript

This last scenario explores the eventuality of coming back to a more static web, without JavaScript. This is the most unlikely today, as it would drastically reduce the dynamicity and comfort of browsing. Yet, there are currently millions of users who have installed the NoScript extension, which gives control to users on which websites JavaScript is allowed to run. We believe that it makes sense to explore the impact of such an evolution on identification through fingerprinting. Currently by disabling JavaScript, some sites do not render at all or render improperly, while most popular sites lose functionality even if properly rendered.

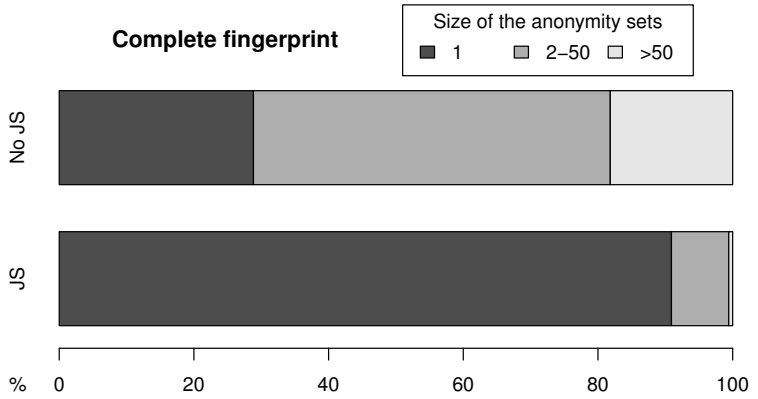


Figure 3.10: Comparison of anonymity set sizes on the complete fingerprint between devices with and without JavaScript

Figure 3.10 shows the impact of the unlikely return to a more static web. The presence of JavaScript in today's web helps make 89.4% of browsers uniquely identifiable, while removing JavaScript reduces the rate down to 29% on our dataset. This percentage could be even lower if user-agents become generic, as stated in scenario n°3. In that case, only 7% of fingerprints would be unique. The privacy benefits are undoubtedly significant but the cost to developers and to the users' comfort would be very high.

Conclusion

Here we have quantified the impact of possible technology evolution scenarii. While some of them could become reality in the not-so-distant future, others are less plausible. Yet, we demonstrate that they can benefit privacy with a limited impact on the beauty of current web browsing.

It is important to notice that tools already exist that can mitigate browser fingerprinting in similar ways as the scenarii discussed in this section. Ad and script blockers, like Ghostery [96]

or Privacy Badger [144], prevent known fingerprinting scripts from being executed in the browser. The NoScript [126] extension blocks the execution of unwanted JavaScript scripts, which is a direct reflection of scenario n°6. The Tor browser team has modified Firefox to create a large range of defenses against browser fingerprinting [181]: from the complete removal of plugins to canvas image extraction blocking, their most recent addition being a defense against font enumeration by bundling a set of default fonts with the browser [182]. This protection illustrates scenario n°5 where the set of exposed fonts is greatly reduced.

3.6 Investigating fingerprint evolution through time

During its lifetime, a device and its components are continually updated to provide new features to the user and to patch security vulnerabilities. From the browser to the the operating system to the drivers, all layers of a system are affected and a single update can have a direct impact on the exhibited fingerprint. Understanding how browsers fingerprints evolve through time is a key concept for both third parties and developers. A third party will be unable to track a device through time if it cannot anticipate correctly possible device evolutions. If two fingerprints present differences, does it come from a device which updated itself or does it come from a completely different device? Data is required to get an insight into this question as a wrong assumption can completely break a tracking system. For developers, understanding the intricacies of fingerprint evolution enables them to design effective defence solutions. They can evaluate if a specific approach is suitable or not depending on how attributes change with time. The content of this section comes from a work in progress on the linkability of browser fingerprints.

3.6.1 Collection of data

We launched in October 2015 an AmIUnique browser extension for Chrome [24] and Firefox [25]. At the time of writing, 767 users have it installed on Chrome and 551 on Firefox. In 20 months, we collected 79,994 fingerprints from 3,528 different devices. When the extension is installed in a browser, it sends a complete fingerprint to the AmIUnique server every 4 hours. If a change is detected, the server will store a new version of the device's fingerprint. If no changes are detected, it will register in the database that the browser has checked in. The storage of simple checks is crucial as it enables us to know if a user has removed the extension or not. The extension also provides a timeline feature to all users where they can see every single modification of their browser fingerprints traced out in an easy-to-navigate representation.

3.6.2 Statistics

Evolution triggers

Browser fingerprints naturally evolve due to several reasons, but each transition can be categorized in one of the following categories:

- Automatic transitions happen automatically without any user intervention. This is mostly caused by automatic software upgrades, such as the upgrade of a browser or a plugin that may impact the `user agent` or the list of `plugins`.
- Context-dependent transitions are caused by a change in the user's real life context. Some attributes, such as `resolution` or `timezone` are indirectly impacted by a change in the environment, such as connecting a computer to an external screen for the resolution or travelling to a different timezone.
- User-triggered transitions require an action from the user. They usually concern configuration specific attributes, such as `cookies`, `Do Not Track` or `local storage`.

As a matter of illustration, Table 3.4 reports the number of transitions per attribute observed between two consecutive fingerprints from the same device. In particular, we observe that the `User agent` attribute is the one with the most changes. Even if it could be considered as unstable, it is

not totally true as the user agent is systematically impacted by software updates with new browser versions. Changes in the user agent are then predictable as most modern browsers follow a regular release cycle. On the lower end, transitions tend to infrequently impact attributes like `cookies`, `local storage` and `do not track`. Interestingly, we also observed some unexpected evolutions of attributes, like `platform`, which are likely to be constant over time. Such transitions reflect that some users of our extension use spoofing mechanisms to artificially modify the content of their fingerprint.

Table 3.4: Number of changes per attribute (fingerprints collected between October 2015 and May 2017)

Attribute	Trigger	Transitions
`User agent`	Automatic	26,199
`Resolution`	Context	19,651
`Platform`	Automatic	10,452
`Plugins`	Automatic	10,213
`Language`	User	9,231
`Canvas`	Automatic	8,422
`Accept`	Automatic	7,176
`Headers`	Automatic	4,423
`Timezone`	Context	4,356
`Encoding`	Automatic	2,798
`Do not track`	User	1,623
`Local storage`	User	893
`Cookies`	User	180

Evolution frequency

Another key indicator to observe is the elapsed time (E_t) before a transition occurs in a fingerprint. Figure 3.11 depicts the cumulative distribution function of E_t. After one day, at least one transition occurs in 58.3 % of the observed fingerprints. The 90^{th} percentile is observed after 9 days and the 95^{th} percentile after 16.8 days. This means that the probability that a transition occurs every 9 days is 0.9, which is non-negligible.

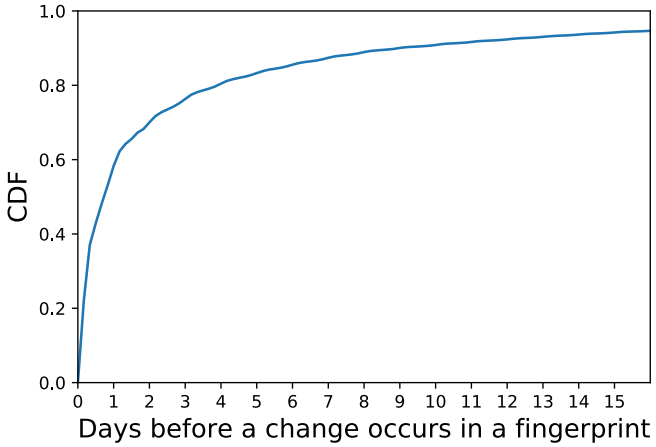


Figure 3.11: Cumulative distribution function of the elapsed time before a fingerprint evolution

In this context, keeping pace with the frequency of change is likely a challenge for browser fingerprint linking algorithms. It remains to be explored if some devices are more impacted by these changes than others. For example, users who connect their laptops to an external screen

at work may see daily changes to their fingerprints even though they did not update a single component of their device.

Evolution rules

While it is difficult to anticipate browser fingerprint evolutions, one can still observe how individual attributes evolve over time. In particular, evolutions of the `User agent` attribute are often tied to upgrading the browser, while evolutions of the `Plugins` attribute refers to the addition, deletion or upgrade of a plugin, which would change its version. Nevertheless, this does not hold for all the attributes, some values might be difficult to anticipate. For example, the value of the `canvas` attribute is the result of an image rendered by the browser instance. The same applies for the screen `resolution`, which can take unexpected values depending on the connected screen.

Based on this observation, the accuracy of linking browser fingerprint evolutions over time requires inferring such evolution rules. To design a good linking scheme, one must take into account both the predictable and unpredictable nature of browser fingerprints.

3.7 Informing users and helping developers

While the primary goal of the AmIUnique website is to collect genuine browser fingerprints, its impact has extended to users and even developers on the web.

3.7.1 Users

One objective with AmIUnique is to inform the public about browser fingerprinting and the underlying privacy implications. We provide on the website the "FAQ" and "Links" sections that detail what we collect, how we collect it and how it can be used with references to articles and scientific publications on the subject. If users want to get a deeper understanding of fingerprinting, they can install a browser extension (see previous Section) that will help them see how their own fingerprint evolve through time. Finally, through the "Privacy tools", we recommend tools so that users can improve their online privacy with browser extensions, privacy-preserving search engines or VPNs.

3.7.2 Fingerprint Central

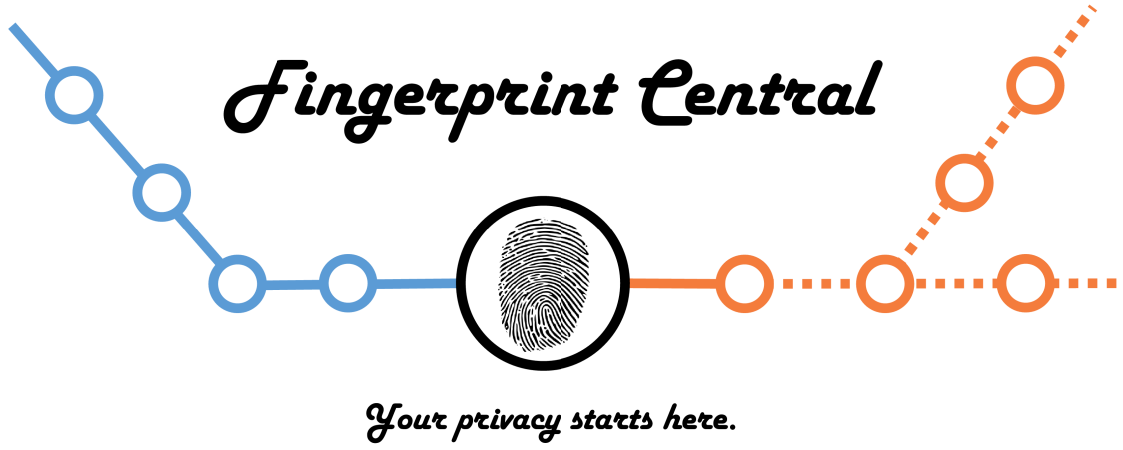


Figure 3.12: Logo of the Fingerprint Central website

As we saw in Section 2.4.2, the approach chosen by the Tor Browser is to present the same fingerprint to all websites but the frequent updates of web browsers pose great challenges. Developers of the Tor Browser must be continually alert about possible changes to existing APIs. They must also watch closely what is released in the literature or on the web to fix any potential leaks

of information. To contribute to the efforts made by the Tor organization, we developed the Fingerprint Central website as part of the Google Summer of Code 2016 [93]. FP Central is a clone of the AmIUnique website but modified to suit the needs of the Tor organisation. When Tor Browser users connect to the website, their browser fingerprint is collected. If the values of some attributes differ from the most popular ones, users are redirected to detailed instructions so that they can get back to the one fingerprint shared by most users. One of the strength of FP Central is that the back-end has been built to address the limitations of the current AmIUnique architecture. With the use of a NoSQL database like MongoDB, it is extremely easy to add new tests to the website or tweak existing ones. Moreover, a customizable statistics page has been added so that developers can easily check what they want in very few clicks. FP Central is expected to integrate the Quality Assurance process of the Tor Browser as reported by this ticket in the Tor Bug Tracker [187]. The complete source code can be found on GitHub at `https://github.com/plaperdr/fp-central` and a beta was deployed at `https://fpcentral.irisa.fr/`.

3.8 Conclusion

To conclude this chapter, the impact of the AmIUnique.org website has far exceeded our expectations. When designing it, we planned on collecting several thousand fingerprints to fuel our research but the number of visitors has been incredible since launch. The number of weekly visitors keeps increasing with no real sign of fatigue and more and more people are installing our browser extensions. The immediate result of this reception is that it gave us the resources to investigate the current state of browser fingerprinting. With the analysis of 118,934 fingerprints, we confirmed Eckersley's findings in 2010 and found that new attributes contribute to the identification of devices on the web. Notably, canvas fingerprinting which relies on both the software and the hardware of a device placed itself in the top 5 of the most revealing attributes in a fingerprint. We also witnessed a shift in existing attributes due to recent browser changes. Plugins which have been deprecated from modern browsers have seen their effectiveness brought to a halt because of security and stability concerns. We also brought evidence that mobile fingerprinting is a reality even if it is for different reasons than on desktops. Despite the constrained diversity of smartphones in terms of software and hardware, identification is still possible because of revealing user-agents and strong canvas tests. As we will see in Chapter 4, the AmIUnique data also gave us the necessary understanding to design smart countermeasures as we were able to see the shortcomings of current defence solutions. By analysing special and strange fingerprints, we were able to see the different mistakes that must be avoided to provide an appropriate protection against tracking. Our authentication scheme detailed in Chapter 5 was also made possible thanks to the data collected from the hundreds of users of our browser extension. By collecting canvas renderings on long periods of time, we are able to prove that canvas fingerprinting is stable enough to harden authentication online.

Outside of the research area, AmIUnique now benefits from a very good visibility. People reference the website on their blog or on social networks when they talk about online tracking and when they want to show what transpire through a web browser. The AmIUnique website has also been included in the official Tor Design document [184] and one reader of the Tor blog even thinks AmIUnique could be run by employees of the FBI or the CIA [183]. This shows that AmIUnique has made its mark on the browser fingerprinting domain since it was launched at the end of 2014.

All in all, the AmIUnique website completely shattered our initial objective and went a lot further than even our most optimistic estimations. It formed the foundations of our research for the past 3 years and it will continue to do so in the years ahead. We are currently planning the next step in the AmIUnique adventure as we want to open the doors to researchers so that it will drive the development of new and promising applications of browser fingerprinting.

Chapter 4

Designing smart countermeasures

As a domain, browser fingerprinting is continually evolving as it relies on the technology embedded in modern browsers. Attributes contained in a fingerprint paint an image of a device at a certain point in time but this image is never finished because its contours are continually redrawn. New APIs are introduced, existing ones are modified and some are even removed.

Designing a countermeasure against fingerprinting is a challenging process as it must take into account both the changing nature of browser fingerprints and the dependencies between attributes. The story told by a browser fingerprint must be flawless and the slightest mismatch between attributes can render a user more visible to trackers. Moreover, the correct balance between privacy and usability is important as the best protection becomes useless if it sacrifices the entire user experience.

In this chapter, we detail two novel defence solutions operating at different layers of a system. We first list in Section 4.1 the key properties that a countermeasure should follow to provide protection against browser fingerprinting. We also highlight the shortcomings of current solutions with regards to these principles. Then, we introduce our first contribution called Blink in Section 4.2. It operates at the OS layer and generates random but genuine browser fingerprints at runtime. The second contribution called FPRandom is explained in Section 4.3. It introduces random noise into very specific fingerprinting routines of a web browser to make some attributes unstable for tracking. Finally, we provide a conclusion to this chapter in Section 4.4.

4.1 Key properties to mitigate browser fingerprinting

4.1.1 Generic properties

A countermeasure is a solution that mitigates the effects of browser fingerprinting. In order to provide protection against identification or unwanted tracking, it should fulfill as many of the following properties as possible:

- **The proposed solution should not break browsing.** As the information contained in a fingerprint represents a snapshot of a device and its configuration, modifying it incorrectly can seriously degrade the user experience to the point where the user cannot even browse the web. For example, modifying arbitrarily the screen resolution can produce an unadapted layout of a web page on the user's device. Changing the HTTP Language header like FP-Block proposed by Torres et al. [185] can result in a web page displayed in a language that the user does not understand. Modifying the user-agent can even prevent the user from downloading the correct version of a software as most websites use this header to guide the user towards the right version. Session breakage is a real concern and developers must restrict their scope of action to preserve the user experience.

- **The proposed solution should not introduce inconsistencies.** The same piece of information can be detected from many different attributes. For example, the platform on which the browser is running can be collected directly from the user-agent and the navigator object.

If a defence solution changes the header but forgets to modify the JavaScript object, it creates a coverage problem as both values will be in contradiction with each other. A countermeasure should not introduce inconsistencies and it should provide a complete coverage of all collected attributes. This way, no third party can detect that a protection is running. However, the constant updates of web browsers makes this property very difficult to maintain. A simple change in a browser API can unmask what the browser was hiding. As we saw in Section 2.4, many defence solutions fall short of that objective as they did not account for all the means to collect the exact same information. For example, the PriVaricator solution tried to prevent font probing through JavaScript by lying on HTML offset measurements [123]. Englehardt et al. showed that this protection could be bypassed by using the *measureText* method of the canvas API [77].

- **The proposed solution should not be detectable.** No third party should be able to create profiles of devices with and without anti-fingerprinting tools. This property is directly linked to the previous one as a fingerprint inconsistency lets a tracker know that a spoofing mechanism is currently running on a device. Time constraints should also be taken into account. If a countermeasure does not introduce inconsistencies in a fingerprint, it can still be identified if it presents a strange temporal behaviour. For example, if a defence solution changes half of the collected attributes between page loads, users of this particular solution can easily be singled out. Finally, no defence solutions should add new artefacts on top of existing attributes (like adding a random number in the user-agent for each request) because they could be used as beacons to identify users on the web.

- **The proposed solution should work automatically without requiring user interaction.** In order to provide the best user experience, the use of a defence solution should be automatic without the need for the user to program complex procedures.

All in all, one must have these properties in mind when designing a countermeasure as they can improve both privacy and usability if they are integrated right from the start. As we will see in Section 4.2, we address with Blink the problem of inconsistencies by generating genuine browsing environments from real running components. In Section 4.3, we explore a new direction to create fingerprint instability and we implement a special strategy in the FPRandom prototype to prevent the detectability of our defence solution.

4.1.2 Design choice in this thesis

Since the diversity of modern devices is one of the root causes of browser fingerprinting, one can ponder if this same diversity can be used as a strategy to prevent tracking. Instead of forcing homogeneity between users like with the Tor browser, we decided to explore what could be done by increasing the diversity of exhibited fingerprints. The intuition behind our solutions is that a browser would present a constantly changing set of values instead of a predictable and reliable fingerprint. This way, third parties would be unable to keep track of a single device as stability would be broken. It would also fill current databases with short-lived fingerprints, rendering the identification process even harder for tracking companies. Because of our choice to increase diversity, a new property needs to be followed to provide protection.

- **The proposed solution should break fingerprint linkability at least between sessions.** The notion of stability is key when designing a countermeasure. Modifying the attributes of a fingerprint is great but if they are kept for several days or weeks, they can still be used to identify a device. A defence solution should present a new fingerprint every time the user wants to browse the web. This way, no third party would be able to link the different browsing sessions of a user. If it is possible, the approach should even try to modify a fingerprint inside the same browsing session to provide maximum protection against tracking. However, as we will see in Section 4.3, it should not be at the cost of detectability.

4.2 Multi-level reconfiguration and diversification with Blink

The diversity of software components to customize browsers is at the source of browser fingerprinting. We argue in this section that this same diversity, combined with multi-level software reconfiguration, provides the foundations for a counter measure to browser fingerprint tracking. The idea is as follows. A browsing platform is assembled by randomly selecting a coherent set of components (an OS, a browser, plugins, etc.). Third parties collect, via the browser, enough information about the platform's components to form a fingerprint. We then regularly reconfigure the platform—thanks to a large set of diverse components—causing a different fingerprint to be exhibited. Our approach breaks one essential property for the exploitation of browser fingerprints: their stability. We propose an original application of dynamic software reconfiguration techniques to establish a *moving target defence against browser fingerprint tracking.*

4.2.1 Approach

This section establishes the threat model we target and goes into the details of our moving target approach. We explain how we leverage software diversity to change a user's browsing platform over time and we discuss the impact this has on both fingerprints and user comfort.

Threat model

We aim at mitigating the exploitation of browser fingerprints to track users, which is a direct threat to privacy. Browser fingerprint tracking relies on the following: web browsers allow remote servers to discover sufficient information about a user's platform to create a digital fingerprint that uniquely identifies the platform. We argue that fingerprint uniqueness and stability are the key threats to browser fingerprint tracking, and in this work **we aim at breaking fingerprint stability over time**.

A moving target defence against tracking

We propose to automatically reconfigure a user's platform to exhibit different fingerprints over time that cannot easily be linked to one another. Figure 4.1 shows the elements of a browsing platform that affect the fingerprint: configuration data at different levels (HW, OS, browser); software components that are assembled at different levels (e.g., `apt-get`, browser plugins, fonts); hardware components, such as the graphics card; cross-level dynamic attributes collectable only at runtime, such as through the HTML5 canvas. Once a user starts browsing the web, these data are used to create a *fingerprint*. We say that a platform *exhibits* said fingerprint.

Our approach reconfigures components that affect the exhibited fingerprint. Nikiforakis et al. [124] show that current commercial fingerprinters collect only a subset of all possible attributes, and Eckersley [72] found the most distinguishing attributes of a fingerprint to be fonts, plugins and user agents. Other attributes are often shared by a large pool of users, rendering them less discriminating. Based on this, we identify the following set of **Diversified Platform Components (DPC)** to be automatically reconfigured: fonts, plugins, browsers, the operating system and the CPU architecture. We call a **DPC configuration** a consistent assembly of a browser running in an operating system, with a specific set of fonts and plugins.

Definition 1. ***Diversity reservoir*** *is the set of components used to assemble new configurations of the DPC. Given a set O of operating systems for different architectures, a set B of browsers of different types and versions, a set F of fonts and a set P of plugins, the reservoir is $DR = O \cup B \cup F \cup P$.*

Our intuition is that the same set of diverse software components that cause fingerprint uniqueness can be exploited to create very large diversification reservoirs. These can be used to build trillions of distinct configurations to switch among. Figure 4.2 illustrates this principle: we randomly select components from the diversity reservoir to create DPC configurations used in the browsing platforms. Over time, we generate new configurations that replace previous ones, changing the browsing platforms and the exhibited fingerprints. The user decides when these changes

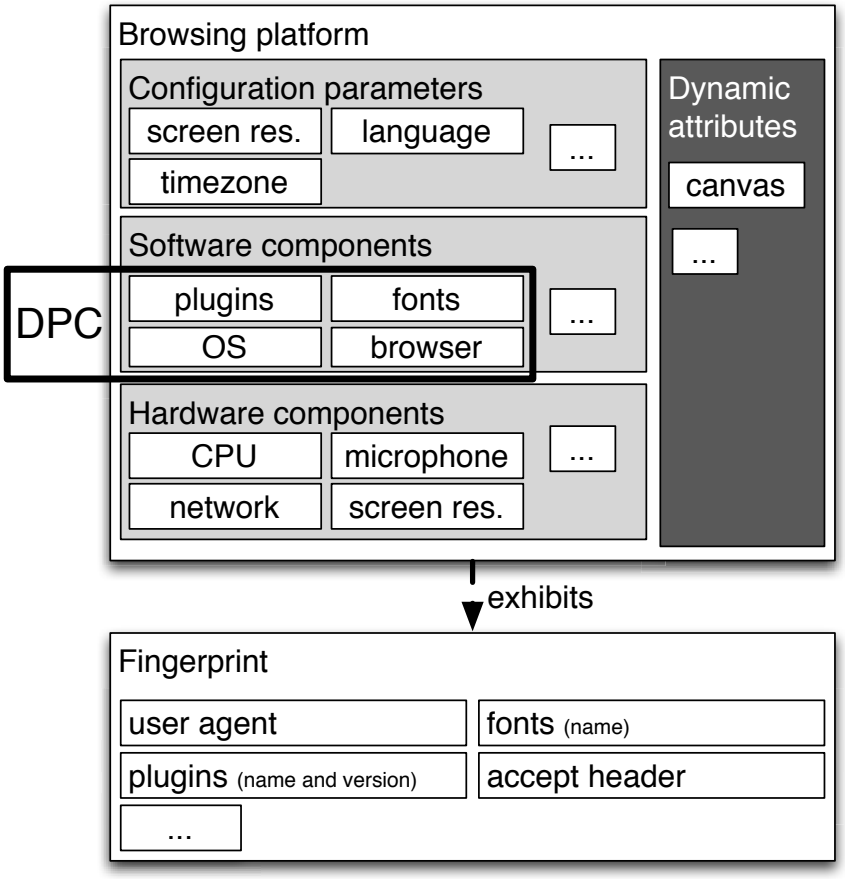


Figure 4.1: User platform elements involved in web browsing and exhibited in the browser fingerprint

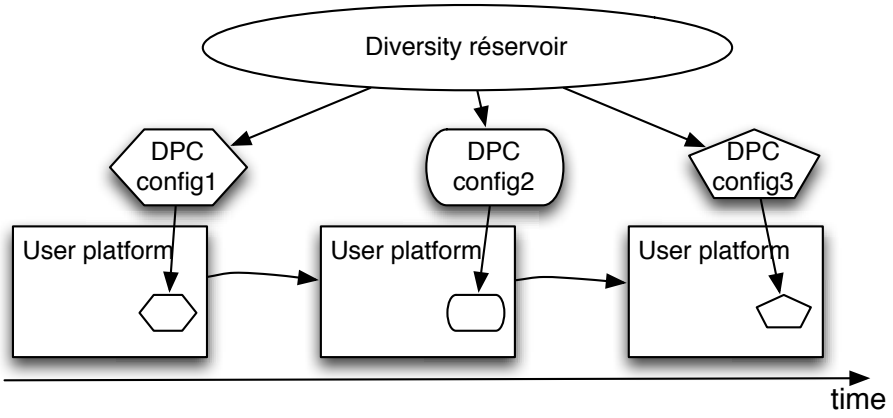


Figure 4.2: Evolution of the user's platform over time

occur. This approach falls into the family of dynamic platforms, a specific form of moving target approaches, as described by Okhravi et al. in [129].

Our moving target defence relies on an essential characteristic needed for reconfiguration: *the modular architecture of systems and browsers*. Modularity makes it possible to reconfigure the browsing platform, on demand, by automatically assembling components. This also allows us to progressively assemble configurations instead of building them beforehand.

Our approach is characterized by three essential properties: (i) the assembled platforms always

exhibit *consistent fingerprints* because the platforms are genuine and we do not lie about any attributes; (ii) we assemble *correct platforms*, i.e. platforms composed of compatible components and which run correctly; and (iii) each *reconfiguration causes the exhibited fingerprints to change*. Section 4.2.2 discusses the model that supports the assembly of correct configuration and in Section 4.2.3, we empirically demonstrate that variations in the exhibited fingerprints deceive two commercial fingerprinting scripts.

Balancing privacy and browsing comfort

There are many places where user comfort conflicts with potential gains in privacy. For example, we have included browsers in the DPC because it is an important distinguishing factor in a fingerprint. Yet, users populate their browsers with data such as bookmarks, passwords and open tabs. We believe that it is important to transfer user data between platforms we assemble in order to keep a comfortable browsing experience despite switching DPC configurations. We developed an offline cross-browser tool that is responsible for transferring the user profile between browsing platforms. We also acknowledge that not all users are ready to randomize their favourite browsers to obtain a different fingerprint, so we let them customize DPC components. Users select components from the DPC that will be included or excluded from all configurations. We call **alterable** and **essential** the components that a user decides to include, respectively discard from the set of DPC.

The choice of an essential or alterable component is subjective and can be done for functional reasons or for comfort. A set of plugins might be crucial for someone's web experience, while another user may simply wish to never change browsers. Furthermore, these choices directly impact the remaining size of the *diversity reservoir* and thus, the search space for randomization. The larger the set of *essential components* is, the smaller the total search space will be.

Deciding when to reconfigure can also impact privacy and comfort. This is a key decision because frequent changes can be effective at mitigating tracking, but are disturbing to users. However, waiting too long between changes increases the time a user's fingerprint remains static, leading to larger sessions that can be tracked. We provide two strategies for reconfiguration, a full reconfiguration strategy, called Leery, and a light-weight one, called Coffee break.

Definition 2. *The **Leery strategy** reconfigures all levels of the browsing platform: the operating system, the browser, the fonts and the plugins. Essential components are kept and alterables are randomized. This strategy is used each time a user starts a new session (i.e. starts a browser for the first time). A fresh DPC configuration is generated and kept until the user ends the session. It draws its name from its cautiousness by reconfiguring as many components as possible.*

Definition 3. *The **"Coffee break" strategy** aims to be faster than Leery by reconfiguring only fonts and plugins. The browser and the operating system do not change. As always, essential components are kept and alterable components are randomized. This strategy is triggered in different ways: manually, by having the user lock his computer, or by waiting for a period of inactivity, hence the name.*

It should be noted that reconfigurations do not occur while the user is browsing the web. To explicitly change the current fingerprint, a new session should be started (e.g., by restarting Blink) or a Coffee break reconfiguration should be triggered. Other strategies are also possible, but we feel these two cover a wide range of uses. The experimental assessment of Section 4.2.5 shows that both strategies can defeat current commercial fingerprinters, but they affect fingerprints differently (more changes lead to more privacy).

Randomizing alterable components

The randomization strategy used to select components at launch is important as an unsuited one could result in the synthesis of uncommon or strange environments. Users would then be detectable as their fingerprints would be a result of a combination of elements rarely found together in the wild. The randomization strategy of Blink cannot follow a uniform distribution as it does not reflect the actual diversity of fingerprints found online. Some plugins or fonts are more common than others and some browsers are more popular than others. To improve the chances that Blink users do not stand out, we rely on real fingerprints collected on AmIUnique.org (see Chapter 3).

We derive empirical distributions from our dataset that dictate which elements are picked and how often they are. This way, the DPC configurations that are generated at runtime are more likely to be found online than those generated from a completely uniform distribution.

Dissimilarity metric

To assess the effectiveness of fingerprint diversification, we need to quantify the differences between fingerprints. To our knowledge, there is no established measure to compare two fingerprints, or to determine if two different fingerprints can be related to a single platform. We define the following dissimilarity metric that aims at capturing the difference between the attribute values observed in two fingerprints.

$$D(FP_1, FP_2) = \frac{\sum_{i=1}^{8} \omega_{attr_i} \times d(attr_i(FP_1), attr_i(FP_2))}{\sum_{i=1}^{8} \omega_{attr_i}}$$

The metric is the sum of dissimilarities between 8 attributes. Its value is defined in the range [0,1]. Each attribute is weighted according to Eckersley's study [72]. Heavier weights indicate more revealing attributes. This captures, for example, that two fingerprints with the same fonts are closer than two fingerprints with the same timezone. We also defined specific dissimilarity functions d for attributes that are lists of values (e.g., fonts, user agent). The weights ω and the dissimilarity functions d are defined in Appendices C.1 and C.2.

To illustrate the intuition behind our metric, let us consider two variations of the fingerprint in Figure 3.1. If we change the browser from Firefox to Chrome and add plugins (as shown in Table 4.1), the dissimilarity between the new and original fingerprints is 0.46. Intuitively, a third party would hardly consider fingerprints with different browsers and plugins to come from the same user. In the second example, only the browser version is modified (Table 4.2). The dissimilarity is then 0.01, which indicates that the two fingerprints are almost identical. This fits our intuition: it is very likely that a browser's version will change at some point in time (e.g., it is updated). Thus, similar fingerprints according to our metric are fingerprints that are likely to be detected as originating from the same user. These two examples illustrate that some differences are more revealing than others, which is what our dissimilarity metric is capturing.

Table 4.1: Changed attributes for example n°1

Attribute	Modified value
User agent	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, Gecko) Chrome/34.0.1847.116 Safari/537.36
Plugins	Plugin 0: Chrome PDF Viewer; libpdf.so, [...] Plugin 12: gecko-mediaplayer 1.0.9;

Table 4.2: Changed attributes for example n°2

Attribute	Modified value
User agent	Mozilla/5.0 (X11; Linux i686; rv:26.0) Gecko/20100101 Firefox/26.0

Positioning w.r.t. existing solutions

Our proposal is to reconfigure the browsing platform at multiple levels, causing the platform to exhibit a different fingerprint each time. It falls into the family of moving target techniques, as described by Okhravi et al. [129]. It is, by their classification, a dynamic platform, and is similar in nature to creating moving attack surfaces for web services [101]. Consequently, Blink's advantage over user-agent spoofers is to never lie. Fingerprints exhibited by the browsing platforms are based on genuine configurations, with genuine browsers, and genuine operating systems. By construction, there are no inconsistencies among the fingerprint attributes, which prevents standing out as a liar. Our approach also relates to secretless strategies because knowing the fingerprint reconfiguration

strategy (random selection of DPC components) is not enough to defeat it. Cox et al. give insight into the wealth of security advantages obtained through software diversity, particularly when there are no secrets to hide [59].

Cross-browser fingerprinting shows that simply changing browsers is not enough because other attributes, such as fonts and plugins, are sufficient to create unique fingerprints [41]. We counter these attacks by randomly changing multiple levels of the browsing platform, not only the browsers.

The main drawbacks of the Tor browser are its usability and the brittleness of its fingerprint. Users should not change any of the Tor browser's configuration options, nor add plugins, because their fingerprint will diverge from the base fingerprint, disrupting the *unique fingerprint* approach and making them identifiable. Our approach is the opposite: we create unstable, always changing platforms that exhibit very different fingerprints. We welcome the addition of user plugins and fonts. And we do not force the use of any restrictive extensions (e.g., NoScript) that severely alter the browsing experience.

Finally, we do not rely on blocking fingerprinting scripts. Maintaining proper lists of scripts requires significant effort. Blink works as a moving target defence system that is oblivious to fingerprinting scripts. Blink also has the potential to resist currently unknown fingerprinting attacks thanks to the use of multi-level reconfigurations.

4.2.2 Implementation

Blink reconfigures the alterable components of a DPC configuration in order to assemble unique browsing platforms. Each unique platform will exhibit a unique fingerprint because the platform components permeate the fingerprint, as seen in Figure 4.1. Hence, by reconfiguring the platform, distinct fingerprints are exhibited. However, several issues need to be addressed for this to work in practice. Namely, the implementation must assemble and reconfigure DPC configurations that lead to platforms that function correctly. The diversity reservoir must be built up to provide a large reconfiguration space. And, finally, changing from one DPC configuration to another should be made as simple and transparent as possible.

This section describes how Blink assembles random browsing platforms, achieves a large amount of fingerprint diversity, and maintains a multi-platform user profile to improve usability. We have implemented Blink using both reconfiguration strategies described in Section 4.2.1. Those implementations are later referred to as **Leery Blink** and **"Coffee break" Blink**.

Multi-level reconfiguration

Blink leverages the modular architecture of operating systems and browsers to randomly assemble browsing platforms that function correctly and exhibit unique fingerprints. To maximize the reconfiguration space and the diversity of exhibited fingerprints, it is important to change as many components as possible, including the CPU architecture, the operating system, the browser, plugins and fonts, which are all statistically important for fingerprinting. Although it is possible to randomize some components directly in the user's system, changes such as adding and removing fonts can have negative side-effects on other applications. More importantly, certain components, such as the operating system or the CPU architecture, cannot be directly changed. It has been shown by Boda et al. [41] and then by Cao et al. [48] that if enough data is gathered from a system it can still be fingerprinted and tracked despite the use of multiple browsers. For these reasons, Blink uses virtual machines to maximize the diversity space that can be exploited by randomizing the operating system and the CPU architecture, all the while maintaining a high degree of isolation between the user's system, fingerprinters and the reconfiguration process.

Blink assembles components at multiple levels to form the browsing platform shown in Figure 4.3. We use the term *multi-level reconfiguration* because the selection of higher level components in the platform directly depends on lower level ones (e.g., a browser plugin depends on the browser, which itself depends on the operating system). Fonts are a special case because although they do not have hard dependencies to or from other components in a DPC configuration, they do impact all user interface applications, including browsers and websites. Moreover, the modularity mechanisms are not the same for all levels: there are different dependency management tools in

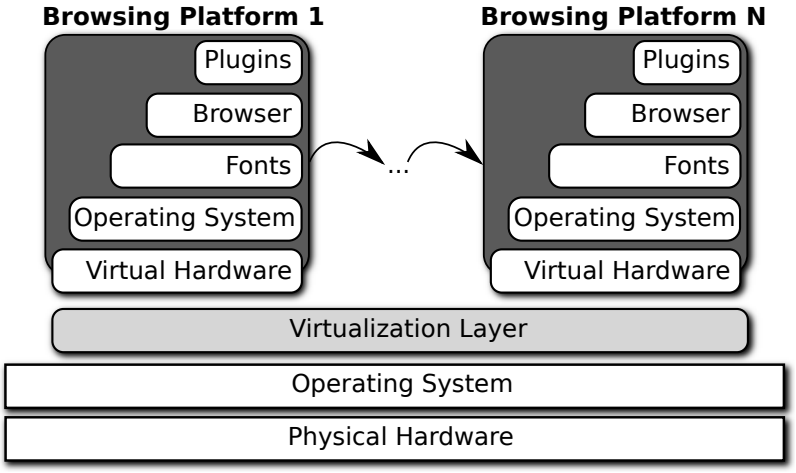


Figure 4.3: A multi-level view of browsing platforms. Virtualization isolates the user's system.

systems and in browsers, different plugin types and versions for different browsers, different CPU architectures, etc. Fonts may also be specific to a given system due to licensing or packaging issues.

1. **Operating system reconfiguration** Blink uses VirtualBox to isolate browsing platforms because it is open source and multi-platform (supports Linux, Mac OS and Windows as both host and guest). Also, VirtualBox abstracts hardware (enabling us to randomize the CPU architecture), provides extensive functionality, has low overhead, and allows for a number of configuration parameters. Added bonuses are a user-friendly GUI and guest additions to improve integration. However, it could be substituted with other solutions.

   We use VirtualBox's shared folders to transfer data between the host and the VMs. Launching a browsing platform starts the assembly of a randomized DPC configuration (i.e. OS, fonts, plugins, browser). The browser, fonts and plugins are copied to shared folders, as is the user's profile (i.e. open tabs, bookmarks, downloads, saved passwords). VMs start the `BlinkVM.py` monitoring script, whose job is to install fonts and plugins, prepare the user profile, launch and monitor the browser, and listen for commands from the host (e.g., shutdown, coffee break reconfiguration). When shutting down, shared folders are cleaned and the user-profile is recovered. VMs are additionally snapshotted when launched and rolled back to their pre-browsing states when the browsing session ends. This ensures the platform returns to a known stable state that is free of temporary files, cookies, trackers, and malware. Finally, because the OS is low-level, reconfiguring it can cause the browser and plugins to be invalidated.

2. **Font reconfiguration** To install a font, the font file must be copied to the correct folder and registered. However, many applications do not detect font changes dynamically. We tested two methods to list the fonts available in browsers, namely Adobe Flash font enumeration and JavaScript font probing. Our results indicate that to ensure font changes are reflected in the exhibited fingerprint you should restart the browser.

3. **Browser reconfiguration** Installing a browser requires copying the installation folder and running the main executable. Browser's run as processes so changing from one browser to another is as simple as stopping and starting them.

4. **Plugin reconfiguration** Installing a plugin is as simple as copying a file to the proper plugin directory. However, unlike fonts, plugins are compiled for a specific CPU architecture and operating system, and can have runtime dependencies on configuration files or system libraries. When reconfiguring, plugins must match the CPU architecture (e.g., i686, IA-64, amd64), the operating system (e.g., Windows, Linux), and the browser (e.g., Firefox, Chrome) of the DPC to work properly. Interestingly, plugin support is often dynamic, as is the case with Chrome, Opera and Firefox. This allows Blink to reconfigure the plugin list at

runtime, and is used for Coffee break reconfigurations to immediately change the exhibited fingerprint without restarting the browsing platform.

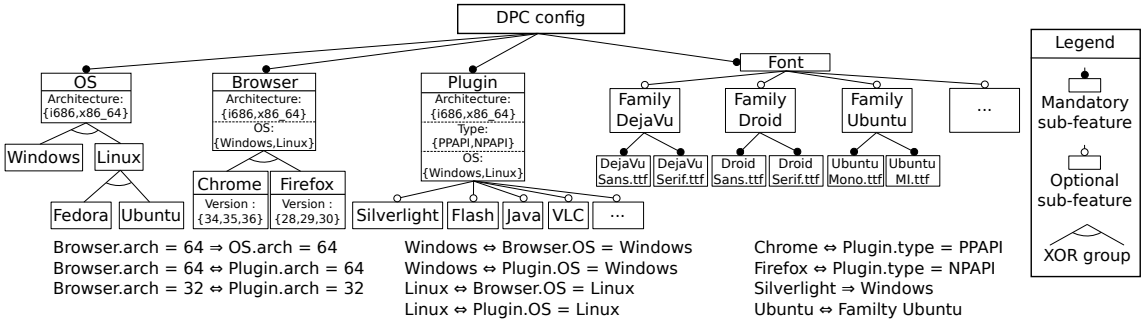


Figure 4.4: An extract of the feature model used for assembling valid DPC configurations

The diversity reservoir

At the heart of Blink is the idea that exploiting software diversity can create an ever changing browsing platform that limits attempts to track its users through browser fingerprint tracking. However, this diversity has to be built; software components must be collected and managed to ensure consistent, functional platforms are assembled and reconfigured.

1. **Building the diversity reservoir** The operating system is the base of the browsing platform. We built VMs using various operating systems, although we focused on the Ubuntu and Fedora Linuxes for the i686 and x86_64 architectures. Many pre-built Linux and *BSD VMs can be downloaded freely over the internet to avoid the installation process. Other operating system's often require licenses and cannot be freely distributed. Once built, each VM must be configured to work with Blink. This includes setting up shared folders and configuring Blink's monitoring script to auto-start.

   We focused on the Chrome and Firefox browsers and downloaded multiple versions of each from the vendors' websites. There's one version for each combination of release channel (stable, beta and development) and CPU architecture (i686 and x86_64). It is easy to add browsers and there are a number of forks for both Chrome (Chromium, SRWare Iron, Epic, ...) and Firefox (Iceweasel, Seamonkey, ...) that are trivial to integrate. To add a browser you must unpackage it into Blink's browser folder and follow the naming convention.

   We wrote scripts to crawl the Ubuntu and Fedora repositories for fonts. We found many duplicate fonts, yet some fonts tend to be specific to a distribution. Moreover, fonts are most often installed in groups, as seen with packages that install multiple fonts. Blink allows defining font groups and system-specific fonts to avoid exhibiting uncommon fingerprints.

   The biggest challenge with plugins is to account for the wealth of browser, operating system and CPU architecture combinations necessary for a plugin to function correctly in each possible DPC configuration. To obtain a large set of plugins for fingerprint diversity, we wrote scripts to crawl the Fedora and Ubuntu Linux repositories and look for plugins in all packages, for both i686 and x86_64 architectures. When a plugin is found, the package is downloaded, and the plugin is extracted. Any dependencies towards, for example, configuration files or system libraries are also extracted.

2. **Modeling the well-formedness constraints in the diversity reservoir** We use feature modeling to ensure DPC configurations are valid before the browsing platforms are assembled or reconfigured. This helps establish the restrictions and conflicts among the different components in the diversity reservoir. The attributed feature model defines the valid configurations, i.e. configurations that can fundamentally run. We provide an extract of the feature model in Figure 4.4. A DPC configuration is composed of 4 mandatory features: an

OS, a browser, plugins and fonts. It must be noted that some features in this model have attributes that specify the feature's characteristics and have domains that define possible values. For example, the attribute *Type* in the feature *Plugin* specifies that a plugin is either of type Netscape Plugin API (NPAPI) or Pepper Plugin API (PPAPI). Relations in the feature model and cross-tree constraints (see bottom of Figure 4.4) specify the set of rules that must be fulfilled for a DPC configuration to be valid and "runnable". For example, the first constraint states that a browser with a 64 bit architecture implies a 64 bit OS.

4.2.3 Research questions

We present a series of experiments to validate the effectiveness of multi-level platform reconfigurations at breaking fingerprint stability over time. The experiments aim at answering the following questions.

RQ1. How diverse is the sequence of fingerprints exhibited by the assembled platforms? This question evaluates the ability of our approach at using the diversity reservoir to assemble series of platforms that exhibit different fingerprints. We measure the dissimilarity between the fingerprints exhibited by each consecutive pair of platforms.

RQ2. How diverse are the platforms in the eyes of actual fingerprinters? This question evaluates the ability of our approach to deceive commercial fingerprinting scripts.

4.2.4 Experiment setup

Our fingerprinting script

All experiments require collecting and analyzing the browser fingerprints exhibited by the assembled browsing platforms. We developed a fingerprinting script inspired by Panopticlick [72], with some improvements to gather more data via JavaScript and Flash. It works by gathering data through HTTP headers and JavaScript attributes, and it uses Flash to gain access to unique attributes in the ActionScript API. We collect some of the attributes listed in Table 3.1.

Third-party fingerprinting scripts

BlueCava is a company that offers websites to "*recognize devices (i.e., computers, mobile phones & tablets)*" for advertising purposes [39]. One tool in their arsenal is a script that uses JavaScript and Flash to fingerprint devices. BlueCava provides a page for users to opt-out of tracking that shows the identifier associated with your device [40]. We collect these identifiers for our assembled browsing platforms. It should be noted that the identification algorithm is on the server side and is unknown to us.

Acar et al. [13] discovered a canvas fingerprinting script on the AddThis website that has since been removed [17]. The script works by sending the browser some text to render using the HTML canvas element. The text is converted into an image that is sent back to the server. Pixel variations in the image indicate differences in hardware and software used in rendering, opening the door to fingerprinting. We use the *getDataUrl()* function of the canvas element to get a URL representation of the image that allows easy comparisons.

Dataset

An **original platform** represents the user's browsing platform as it exists in the host operating system. The content of this platform can have a strong impact on Blink's ability at producing diverse platforms, since we import the user's fonts and plugins when assembling a DPC configuration. In essence, the plugins and fonts of the user's platform are included as essential components in the assembled platforms.

We test Blink's effectiveness using 25 original platforms. Each original platform has from 1 to 25 plugins, and from 20 to 520 fonts. The diversity reservoir for our experiments is: 4 Virtual machines (Fedora 20 32/64 bits, Ubuntu 14.04 32/64 bits); 2762 fonts; 39 browser plugins; 6 browsers, including 3 versions of Firefox: 28.0 (stable), 29.0 (beta), 30.0 (aurora), and 3 versions of Chrome: 34 (stable), 35 (beta), 36 (unstable).

Experimental protocol

For each of the 25 original platforms, we assemble 2 sequences of 100 platforms, providing a total of 5000. For every assembled platform, we collect the exhibited fingerprint using our script, and we collect the BlueCava and AddThis identifiers using their commercial fingerprinting scripts.

4.2.5 Results

RQ1. How diverse is the sequence of fingerprints exhibited by the assembled platforms?

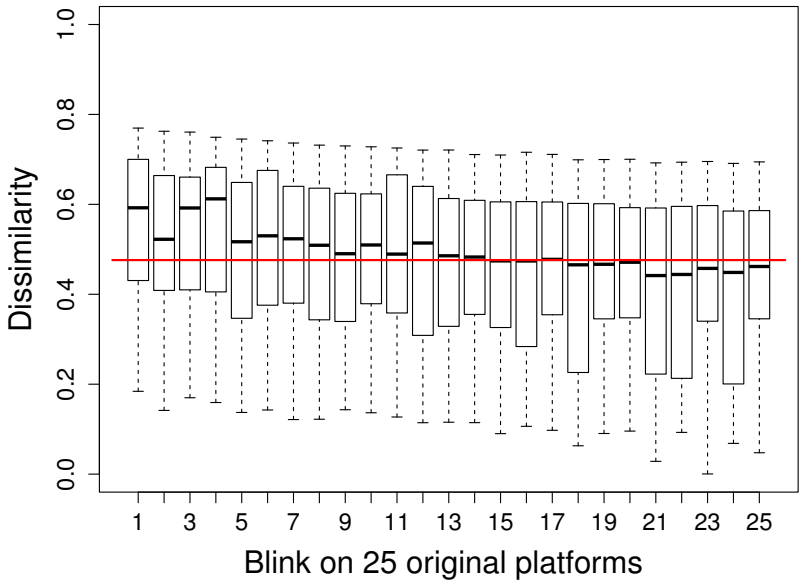


Figure 4.5: Dissimilarity between consecutive platforms (Leery mode)

This question aims at validating that the browsing platforms we assemble exhibit different fingerprints over time. We estimate this by checking that two consecutive platforms exhibit fingerprints that are dissimilar. Thus, for each original platform, we collect the sequence of fingerprint dissimilarities by pairs of consecutive platforms:

$dissim_seq = (D(FP_i, FP_{i+1}))_{1 \leq i \leq 99}$

For each of the 25 original platforms, we collect two sequences of dissimilarity values, which correspond to the 2 sequences of 100 assembled platforms. Figure 4.5 displays the distribution of dissimilarities in both $dissim_seq$ for each original platform, in Leery mode. Figure 4.6 displays them in Coffee break mode. The X-axis in the figures correspond to the ID of the original platform, and the Y-axis represents the distribution of dissimilarities between pairs of consecutive platforms. The red line indicates the mean dissimilarity value among the 5000 collected fingerprints.

Regarding the Leery strategy, Figure 4.5 shows that Blink is successful at assembling successive platforms that exhibit very dissimilar fingerprints, with results up to 0.77. Blink can also assemble platforms that are very similar, with some pairs of successive platforms having dissimilarity values as low as 0.01. Since we have a relatively small pool of operating systems and browsers, it is likely that the OS or browser was the same from one configuration to the next. So, depending on randomization, consecutive fingerprints can be very similar or very dissimilar. Yet, despite frequent, large variations in dissimilarities, mean values are high thanks to our large pool of fonts and plugins.

When comparing Figures 4.5 and 4.6, the most noticeable differences are the ranges of dissimilarity. In Leery mode, dissimilarity varies between 0.01 and 0.77, while in Coffee break mode dissimilarity varies between 0.08 and 0.36. This shows the benefits of switching the operating

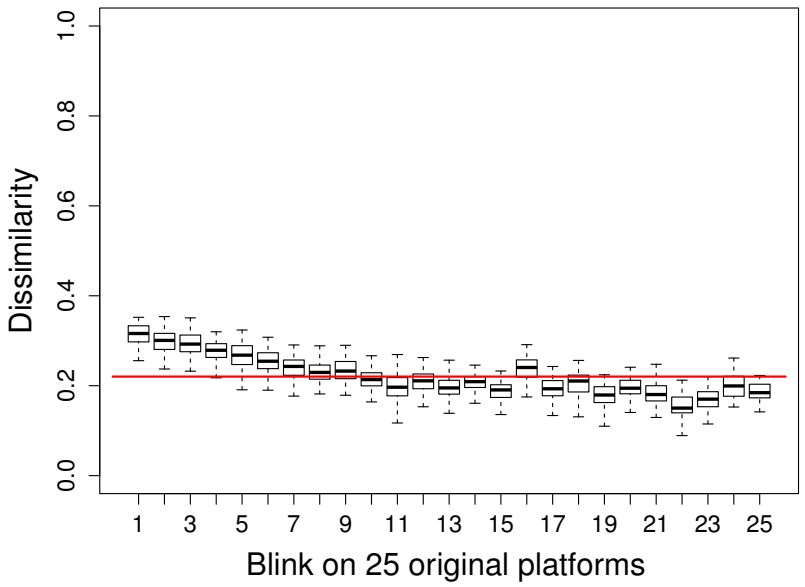


Figure 4.6: Dissimilarity between consecutive platforms (Coffee break mode)

system and browser. The mean dissimilarity values in Leery mode are more than twice as high as the ones in Coffee break mode.

Dissimilarity slowly declines as the numbers of fonts and plugins grow with each original platform. This trend occurs with both reconfiguration strategies and can be explained by the fact that the assembled platforms start to share a larger set of common plugins and fonts, lowering the overall dissimilarity score for both attributes. In the end, even if the dissimilarity is lower if we import a lot of fonts and plugins from the user's system, Blink still produces highly dissimilar configurations with an average above 0.4 in Leery mode.

We conclude from these observations that Blink generates configurations that are truly different from one another. It corresponds to our objective of breaking the stability of fingerprints over time, and shows its effectiveness in blocking the ability of fingerprinters to track users. Variations in Blink's effectiveness are due to the richness of the original platforms and on the size of the diversity reservoir.

RQ2. How diverse are the platforms in the eyes of actual fingerprinters? To answer this, we collected fingerprint identifiers from BlueCava and AddThis for each assembled platform. When these scripts assign the same identifier to two platforms, it means they consider them to be the same.

BlueCava browser fingerprinting script In our tests more than 99% of the identifiers we collected were different (we had similar results for both Leery and Coffee break modes). BlueCava's script is not able to detect that the assembled platforms are from the same user, in part due to BlueCava taking into account even the slightest differences in a fingerprint. This shows that Blink is effective against unique identifiers, and confirms that Blink can deceive BlueCava, even when fingerprint diversity is low.

AddThis canvas fingerprinting script From the 5000 assembled platforms, we obtained 181 different identifiers in Leery mode and 34 in Coffee break mode using the AddThis script. Blink performs relatively well despite the fact that canvas fingerprinting targets the GPU and low-level drivers. We can also see that Leery is 5 times more effective than Coffee break because the OS and browser change. Currently the DPC does not include graphical components (e.g., GPU drivers) or GPU/HTML5 specific configurations, but we plan to explore this in the future.

4.2.6 Threats to validity

To our knowledge, characterizing the impact of moving target defenses on security is an open challenge [51]. Still, this section provided empirical evidence of the effectiveness of Blink's behaviour with respect to fingerprinting. We now summarize the threats to the validity of our findings.

While no established metric exists to evaluate the effectiveness of fingerprinting countermeasures, we defined our own dissimilarity metric. This is a construct threat because the metric might not properly reflect the ability of fingerprinters to decide if a fingerprint is from a distinct platform. To mitigate this threat, we based fingerprint attribute weights on the observations from Eckersley's extensive study [72]. We also collected identifiers from third-party commercial fingerprinting scripts.

The external validity lies in the ability to generalize our observations. We evaluated Blink's effect on 25 initial situations, with a pool of 6 browsers and 4 operating systems, all running Linux platforms. Starting from these situations, Blink is able to assemble platforms that exhibit very different fingerprints. Yet, we do not know how Blink behaves in other situations (e.g., users that have a very large set of essential fonts). We are exploring a new domain (mitigating fingerprint-based tracking) and further quantitative and usability studies are needed to establish a comprehensive understanding of moving target approaches in this domain.

Internal validity very much depends on the correct implementation of Blink, as well as both the fingerprinting scripts and the metrics. We mitigated this risk through thorough testing campaigns, and thousands of runs to tune Blink. Yet, there might be bugs that influence our results. We hope that they only change marginal quantitative things, and not the qualitative essence of our findings.

4.2.7 Discussion and further development

Mitigation properties

Blink fulfills all the properties defined in Section 4.1:

- Blink does not break browsing. It runs like a native browser and we transfer the user profile between different sessions to improve the user experience.
- Blink is completely automatic. The tool has been built so that the synthesis of browsing environment is frictionless and it does not require user interaction.
- Blink does not introduce inconsistencies. The strongest advantage from Blink compared to other tools is that the exhibited fingerprints are genuine with no mismatches between attributes.
- Blink breaks fingerprint linkability between sessions by generating a completely new environment every time it is launched.
- It is very difficult to detect Blink. Users of our approach could potentially be spotted as having "strange" fingerprints. To mitigate this effect, we can rely on real fingerprints collected on AmIUnique.org (see Chapter 3). We observed that fingerprints rarely have more than 20 plugins and that the number of fonts range from 50 to 550. We derived two normal distributions from the data we collected and use them to set the number of fonts and plugins that we add to each DPC configuration. This way, we do not end up with configurations that would never be observed in the wild, improving the chances that Blink users do not stand out.

Improving performance

It can be argued that using virtual machines to run browsers is costly. However, current computers can easily run multiple VMs simultaneously. Many developers already use multiple VMs to target different platforms. VMs allow Blink to target multiple platforms and architectures, and to reconfigure the OS, all important discriminators in fingerprints. Moreover, the use of VMs presents beneficial side-effects: it provides very strong sandboxing for all web activity, including Firefox,

which does not have a native sandbox. In our implementation, snapshots are used to return VMs to known safe-states, removing cookies and other browsing artifacts.

Since the publication of the original article, we developed an alternative version of Blink running completely inside Docker containers. Docker enables developers to have self-contained software by packaging everything in a container [68]. The advantage is that a container only has the libraries and settings required for the application to run. The HDD space is then reduced compared to a fully-fledged virtual machine and the performance of the system is much faster. The complete infrastructure of Blink on Docker is available publicly on GitHub [8].

Relevance

The original version of Blink was developed in 2014. At the time, plugins and fonts were the kings of collected attributes in a fingerprint. Now, as saw in our AmIUnique study, the impact of these attributes are getting extremely limited as modern browsers do not support the old NPAPI plugin architecture anymore. The current implementation of Blink may be less effective at mitigating tracking than at the time it was developed. Yet, the approach chosen when designing Blink is still relevant today. In order to provide a complete coverage of modified attributes, it is better to rely on running components than to artificially modify the content of a fingerprint with pre-recorded values. Mismatches between attributes are prevented as the browsing environment reflected in the fingerprint truly exits.

4.3 Randomizing core browser objects with FPRandom

In this section, we explore the use of browsers' flexibility to prevent tracking through advanced fingerprinting techniques along with an implementation in a Firefox browser called FPRandom. We add randomness in the computation of selected browser functions, in order to have them deliver slightly different answers for each browsing session. This way, these functions are considered unstable for tracking and cannot be used for fingerprinting.

4.3.1 Approach

In this work, we propose to exploit browsers' untapped flexibility to introduce randomness. Several API functions used to build a browser fingerprint are unnecessarily deterministic and provide device-specific information as a side effect. Instead of changing software components at runtime like Blink [1] or lying on specific values like PriVaricator [123], we want to increase non-determinism in browsers to reduce these side-effects that cause fingerprintable behaviours. We especially investigate the two following areas.

Flexibility of the implementation of the JavaScript specifications

The official ECMAScript specification, the de facto standard for scripting language on the web, allows some flexibility in actual JavaScript implementations. Different parts of the specification give some leeway by clearly indicating that specific choices are left for the implementation. The ECMA organization strictly codifies the interpretation of the language but the exact details of how it works remain in the hands of browser vendors. For example, as we will see in the next section, the enumeration order of JavaScript properties are not detailed by the ECMAScript specification but each browser presents its own unique order. Developers have made deterministic choices when they implemented these functions. By taking a step back and removing what can be interpreted as a surspecification of the standard, we are able to thwart fingerprinting vectors that rely on these detectable side-effects.

Flexibility of the renderings of multimedia elements

Vendors are constantly striving to improve their browsers to provide the latest innovations and the best possible experience to their users. Changing some attributes collected in a browser fingerprint like the user agent or the screen resolution can negatively impact how a page is displayed to the

detriment of users. However, the rendering of HTML multimedia elements can be made more flexible and less deterministic without degrading the user experience. Especially, we can exploit users' perception of color and sound to introduce imperceptible noise that impacts the stability of specific browser routines. The key challenge here is to apply very small modifications that no user will notice while a fingerprinting script will output constantly changing values at every execution.

4.3.2 Implementation

To experiment with randomization, we target three of the most recent fingerprinting techniques: canvas fingerprinting as it is a prime example of a dynamic media element and "in the top 5 of the most discriminating attributes" [2]; the Web Audio API recently observed in fingerprinting scripts by Englehardt et al. [77]; the leakage of system information through JavaScript properties' order found by Nikiforakis et al. in the Cookieless Montster study [124]. All the mitigation techniques detailed in this section are implemented in a modified version of Firefox called FPRandom. The complete patch for Firefox 54 is available on GitHub along with a fully-compiled prototype for Linux systems [12].

Canvas API

Definition Canvas fingerprinting was firstly introduced by Mowery et al. [116] and observed on the Internet by Acar et al. [13]. Its goal is to use the Canvas API of a browser to draw an image that can help differentiate one device from another. Each device executes the exact same set of instructions and depending on both hardware and software, rendered images present variations. Figure 4.7 shows the canvas test we run on the AmIUnique website. The test consists in displaying two lines of text with different shapes merged together. Here, depending on the hardware and the installed drivers, the rendering of shapes and colors slightly vary between devices. Then, depending on the software and most especially on the list of installed fonts, the lines of text can present great differences. In our AmIUnique study [2], we showed that one of the strength of canvas fingerprinting is its stability and that it is "in the top 5 of the most discriminating attributes". It is notably the "second highest source of entropy for mobile devices".

Modification The first modification we made to the Firefox source code is to introduce randomness inside the *ParseColor* function of the *CanvasRenderingContext2D* class. Every time a color is set inside a canvas fingerprinting script, the browser changes the actual RGB values of the parsed color by adding or removing a very small number for each color channel. For example, if a script asks to draw an orange rectangle, the browser will paint the canvas element as requested. However, for every browsing session, the browser will use a slightly different orange than the last time. Modifying the *ParseColor* method enables us to support the full range of color declaration (for example, you can chose a color by directly setting its RGB values or you can simply write its name like "gold" or "orange"). The impact on the user experience is almost non-existent as the difference in color is very hard to see with the naked eye. Finally, it should be noted that we differentiate ourselves from tools called "canvas poisoners" that change the RGB values of each pixel of the rendered image independently from one another. Mowery et al. wrote that they are not "a feasible defense" against current scripts because the noise can be lifted by repeating a test a few times and comparing the results. They add that the aggressive introduction of noise "degrades the performance of <canvas> significantly for legitimate applications" [116]. With our approach, the color on a whole surface is consistent as we do not introduce random noise on separate pixels. As discussed in Section 4.3.2, we can apply the exact same modification for every run of a fingerprinting script. If a fingerprinter were to repeat the same canvas test more than once, he will not be able to notice differences whereas canvas poisoners present noticeable variations between runs.

The second modification operates in the *SetFont* function of the *CanvasRenderingContext2D* class and changes a font set by a script by one present on the operating system. For the scripts asking for a fallback font, the stability is broken as a font different from the previous session will be presented.



Figure 4.7: Original canvas rendering with standard colors and the default fallback font



Figure 4.8: Canvas renderings with modified colors and fonts

Example Figure 4.8 illustrates the impact of FPRandom on the exact same canvas test with all protection features enabled. The blue, orange and green colors are slightly different for each run and the used fonts are chosen among the ones present on the operating system. The more fonts are installed on the user's system, the bigger the diversity of generated canvas renderings will be. By changing at runtime core properties of elements present in a canvas test, we break the stability of this technique while still preserving the user experience.

AudioContext API

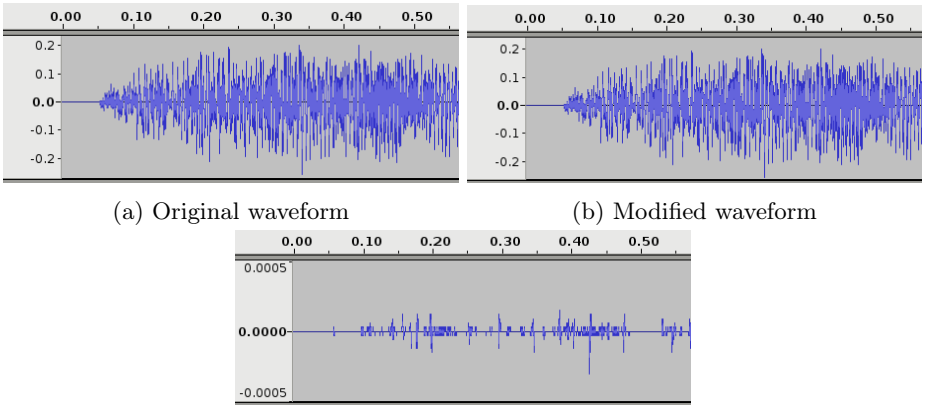
Definition Discovered by Englehardt et al. while crawling the web looking for trackers [77], AudioContext fingerprinting is a newcomer in the browser fingerprinting domain. The AudioContext API provides an interface to create a pipeline to process audio. By linking audio modules together, you can generate audio signals and apply very specific operations like compression or filtering to generate a very specific output.

Listing 4.1 provides an example of an AudioContext fingerprinting test. Lines 1 to 5 create the different modules that are used in the audio pipeline. Lines 8 and 9 set parameters on two different modules. Lines 10 to 13 link the modules together. The complete pipeline of the example is as follows. By starting the *oscillator* node at line 16, the module outputs a triangle wave that is passed to the *analyser* node. The audio continues through the *scriptProcessor* node that allows specific operations in JavaScript. It then goes through the *gain* node before reaching its final destination. The strength of the Web Audio API is that any number of modules can easily be linked together to form a complete audio pipeline.

```
1  var audioCtx = new(window.AudioContext || window.webkitAudioContext),
2      oscillator = audioCtx.createOscillator(),
3      analyser = audioCtx.createAnalyser(),
4      gain = audioCtx.createGain(),
5      scriptProcessor = audioCtx.createScriptProcessor(4096, 1, 1);
6
7
8  gain.gain.value = 0; // Disable volume
9  oscillator.type = "triangle"; // Set oscillator to output triangle wave
10 oscillator.connect(analyser); // Connect oscillator output to analyser input
11 analyser.connect(scriptProcessor); // Connect analyser output to scriptProcessor
       input
12 scriptProcessor.connect(gain); // Connect scriptProcessor output to gain input
13 gain.connect(audioCtx.destination); // Connect gain output to audiocontext
       destination
14
15 scriptProcessor.onaudioprocess = function (bins) {...}
16 oscillator.start(0);
```

Listing 4.1: Excerpt of the AudioContext fingerprinting script found by [77]

In audio, sampling is applied to convert a continuous signal into a discrete one. This way, a computer can easily process audio in distinct blocks called frames. Each frame is composed of samples that represent the value of the audio stream at a specific point in time. Englehardt et al. have shown that, depending on the audio stack of your system (both software and hardware), the exact value of each of these frames slightly vary between devices. An audio fingerprint can then be created similarly to what is done with the Canvas API.



(a) Original waveform

(b) Modified waveform

(c) Difference between the two waveforms

Figure 4.9: Visualization of audio rendered through the AudioContext API

Modification We performed an analysis of audio fingerprints that we collected on AmIUnique.org and the results can be found in Appendix D.1. We decided to introduce very small noises directly into the audio processing routines of the browser so that tests using any number of AudioContext modules are all impacted. We operate at the heart of the *AudioBuffers* of the *AudioNodeEngine* as they contain the frames of the processed audio. By modifying key functions, we slightly decrease the volume of processed buffers by a factor ranging between 0.000 and 0.001. This way, a frame can present very small variations where only the smallest decimal digits are affected. With the use of very small factors, it is impossible to detect modified sections from unmodified ones just by listening to the rendered track as the differences between the original and modified track can genuinely be interpreted as side effects or simple noise of the whole audio stack of the device. For fingerprinting scripts, these modifications produce a different hash as the audio routine will be ever so slightly different for each browsing session.

Example Figure 4.9 shows three waveforms of the first second of the "Ride of the Valkyries" from Wagner. The audio pipeline we set up for this example performs two operations. It first increases the volume of the track with a *GainNode* and then compresses it through a *DynamicsCompressorNode*. The waveform in Figure 4.9a represents the output from an unaltered pipeline and the one in Figure 4.9b from a pipeline with our volume modification. The last waveform in Figure 4.9c represents the difference between the first two (i.e. the introduced noise). In order to see the impact of FPRandom, the 3rd waveform has been zoomed in at 1000%. The scale is a clear indication that the generated noise is inaudible, proving that the impact on the user experience is non-existent audio wise but it still impacts the created audio fingerprint.

Order of JavaScript object's properties

Definition By analyzing the content of JavaScript objects in the browser, Nikiforakis et al. discovered that "the order of property-enumeration of special browser objects, like the navigator and screen objects, is consistently different between browser families, versions of each browser, and, in some cases, among deployments of the same version on different operating systems" [124]. This way, if someone were to hide the true browser's identity, enumerating the properties of a special object would simply unmask it. As stated by the latest ECMAScript Language Specification ratified in June 2016, "mechanics and order of enumerating the properties is not specified" (see section 13.7.5.15 EnumerateObjectProperties of [74]). This ordering behaviour is entirely dependent on the browser's implementation. Chrome and Firefox yield vastly different enumeration orders for native objects like *navigator*. For non-native JavaScript objects, both browsers first return integers in ascending order and then strings in insertion order. This choice is arbitrary and many developers have long debated for the best and most logical behaviour as illustrated by this long discussion on the V8 bug tracker [134].

Modification The browser's unmasking added by the surspecification of the ECMAScript standard can simply be undone by modifying the *jsiter* class of Firefox. A special flag called "JS_MORE_DETERMINISTIC" can be activated at compile time to sort IDs for both native and non-native objects in a deterministic way. By tweaking the behaviour of the *SortComparatorIds* structure used with this special flag, we flip its purpose by not making the JavaScript engine more deterministic but by generating a unique enumeration order every time the browser is launched.

With the "JS_MORE_DETERMINISTIC" flag activated, the enumeration of a JavaScript object first returns integers in ascending order and then strings in alphabetical order. By diving even deeper into the source code, we found that the string comparison done by the browser relies on the "Latin-1" or "ISO/CEI 8859-1" encoding of each string. When comparing two strings, the engine goes through one character at a time and performs a simple subtraction of their code points (i.e. their place in the Latin-1 character set, see [202]) to determine which character is in front of the other. When a difference is detected, the engine knows how to order the two strings as the result is either positive or negative. Appendix D.2 gives an example of such comparison between the *appName* and *appVersion* strings.

In order to change the enumeration order for each browsing session, we assign a random order for each combination (i.e. for each possible subtraction result) from the Latin-1 character set. As the first code point starts at position n°32 and the last one is at n°255, we generate in total 223 different booleans to cover all possible combinations. Any attempt to unmask the browser through this technique is then prevented.

Randomization strategy

All the modifications described in this section can be executed in different ways when browsing the web. Here, we detail the two randomization strategies present in FPRandom while discussing their own strengths and weaknesses.

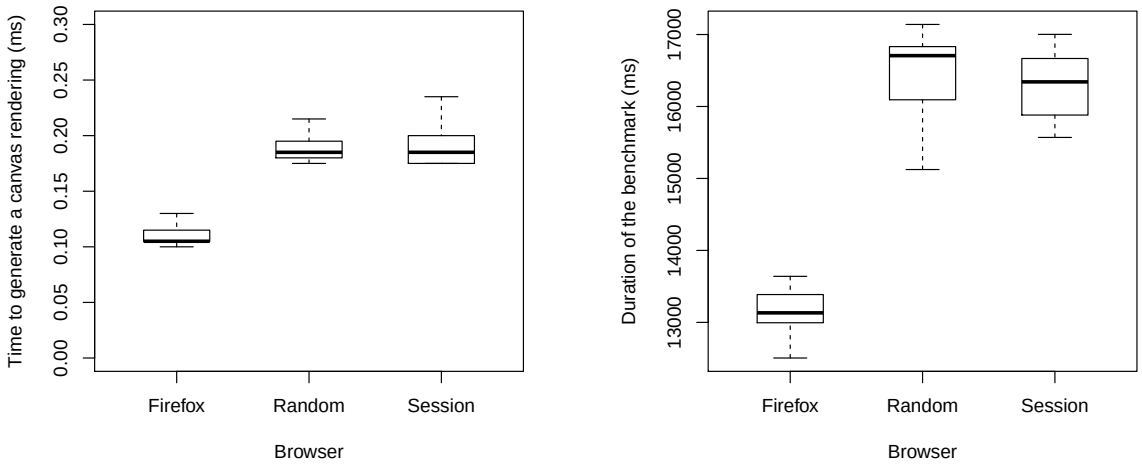
Random mode The first mode that we propose in FPRandom is the "Random" strategy. Every time the modified functions are executed in the browser, they will return random values. The advantage is that it prevents cross-domain tracking as two scripts on two different domains (even from the same provider) would collect different values on both sites. However, the major downside of this solution is that it presents "transparency" problems as discussed by Nikiforakis et al. in the PriVaricator study [123]. If a fingerprinter were to study the presence of randomness, a script could execute the same test several times to detect instability. Depending on the test, a statistical analysis could be performed to reduce or remove the introduced randomness but it requires far more means and a certain insight into the way noise is introduced to get meaningful results. The "Random" mode is the default one in FPRandom as we have no recorded proof of such behaviours from today's major fingerprinting actors.

Per session The second mode initializes all the randomized variables at startup and they are never modified on subsequent executions. The advantages of this strategy is that it cannot be detected through repeated measurements as the browser will always return the same answers for an identical fingerprinting test. The downside is that it only breaks linkability between browsing sessions as the same fingerprint will be presented to all websites until the browser is rebooted.

4.3.3 Evaluation

Deceiving fingerprinting scripts

As pointed out by [123], while it is possible to analyse the JavaScript code that runs inside the browser and detect fingerprinting scripts, it is much more complicated to find fingerprinters that can act as black-box oracles for our work. Some websites give a specific identifier associated with a device's fingerprint but others map collected attributes in a very specific way that is confidential and that is entirely performed on the side of the server. The main challenge in assessing the impact of FPRandom is to find fingerprinting scripts that use the advanced techniques we target



(a) Canvas API (b) Audio API

Figure 4.10: Benchmarking results

and retrieve the right information (either an identifier or the fingerprint data that is sent).

Fingerprintjs2 is the second version of a popular open-source fingerprinting library that collects 25 different attributes and hash them all into a single value [159]. We executed the complete test suite of this library 100 times on both a standard version of Firefox 54 and FPRandom. On Firefox 54, we obtained the same hash for all of the 100 executions. For FPRandom, we collected 100 different ones with the Random mode and a single one in Session mode. These results show how fragile the test suite is for identification. The introduction of noise on a single attribute is sufficient to be considered as a "new" fingerprint.

Maxmind is a company specialized in IP geolocation and online fraud prevention. As part of its fraud detection system, Maxmind has a "device tracking add-on" to identify devices "as they move across networks" [94]. The main add-on script sends the complete device fingerprint at a specific address in a POST request. We manually analysed the code of the add-on and found that it collects the rendering of a canvas test along with the enumeration order of both the *navigator* and *screen* objects. After 100 runs of the fingerprinting script, FPRandom gives a different canvas hash at each execution whereas a standard Firefox build always send the same result. For the enumeration orders, the behaviour of Firefox 54 is the expected one and returns the exact same order for both JavaScript objects. For FPRandom, the browser gives a unique and different enumeration order at each session.

Limitations Our approach does not deal with static attributes like the user-agent or the timezone but it can mitigate the collection of dynamic attributes from APIs like Canvas or Battery. Scripts that do not rely on the attributes we target can still build their own browser fingerprint and use it for tracking, albeit with a less complete view of the user's system.

Performance

We use three different JavaScript benchmark suites to assess the performance overhead introduced by FPRandom. The experiments were conducted on a laptop running Fedora 25 with an Intel Core i7-4600U CPU @ 2.10GHz. The tests were performed using Firefox 54 (Nightly version) with and without our modifications present and enabled.

Canvas As there are no benchmarks that specifically target the Canvas API, we developed our own test to assess the overhead introduced by our color and font variations. We repeated the test shown in Figure 4.7 1,000 times and measured the time it takes for each image to be fully rendered inside the browser. To get precise measurements, we used the JavaScript Performance API that provides timestamps with an accuracy up to the microsecond [180]. Figure 4.10a illustrates the difference

between a vanilla version of Firefox 54 and FPRandom. While an unmodified version takes 0.12ms to render the image, our modified browser is about 0.06ms longer on average in both randomization modes. This difference corresponds to the time it takes to properly choose a new font and introduce variations in the canvas colors. With these reported numbers, we consider the overhead here to be negligible as a rendering time of less than a single millisecond will not impact the user experience.

AudioContext To assess the impact of FPRandom on the AudioContext API, we use a WebAudio benchmark developed by a Mozilla developer [31]. The benchmark performs a series of 19 different tests from simple gain operations to more complex mixing procedures. Each test instantiates an *OfflineAudioContext* object which is one of the objects we targeted when modifying the Firefox source code. The results in Figure 4.10b indicate the time it takes to perform the complete series of 19 tests. It should be noted that we repeated the test suite 30 times. The mean value for FPRandom is about 25% higher than its Firefox counterpart. The "Random" mode is also a little longer than the "Session" one. This increase can be explained by the fact that the modified code is executed a very high number of times in a single test. By comparison, the modification made to the canvas API is only executed once for a test. We instrumented the browser to find a precise number and we found out that a single run of the benchmark enters our modified function more than 8,862,000 times. As a point of reference, the AudioContext test found by Englehardt et al. [77] only enters our function less than 3,000 times. With these numbers, we qualify the benchmark as extremely intensive. The increase in processing time may not be visible for less extreme and more traditional uses of the API. We leave for a future work the exploration of different implementation strategies where only a specific percentage of audio frames would be modified, leading to an increase in performance.

JavaScript enumeration order As a final performance test for FPRandom, we decided to run a standard JavaScript benchmark to see if the modification made on the enumeration order has an impact on the overall JavaScript engine. We used the recent JetStream benchmark [107] which is developed as part of the WebKit browser engine. Currently, the 1.1 version performs 39 different tests and covers a wide variety of advanced workloads and programming techniques. It integrates tests from well-known benchmarking suites like SunSpider or Octane along with new ones developed specifically for JetStream (more details [106]).

Table 4.3: JetStream benchmark results

	Firefox	FPRandom-Random	FPRandom-Session
Latency	76.075 ± 1.3250	74.553 ± 1.8074	74.767 ± 1.2530
Throughput	251.97 ± 3.2912	252.32 ± 2.4214	256.02 ± 1.1213
Total	147.45 ± 1.5753	146.23 ± 1.9204	147.61 ± 1.1257

The results are present in Table 4.3 (the bigger the score, the better the performance). As we can see, the scores are almost identical and no real distinction can be made between Firefox and FPRandom. The behaviour of the two browsers are similar on both JavaScript throughput and latency and the impact of our order modification is seemingly nonexistent.

Web crawl In order to assess more globally the impact of our modifications on day-to-day browsing, we crawled the thousand most popular websites as reported by Alexa [19] on both a vanilla version of Firefox 54 and FPRandom. We used Selenium as the engine for the crawl of both browsers, and we used the Performance API of the browser to measure the time it takes for the DOM of the main page to be completely loaded. Specifically, we used the *domLoading* and *domContentLoadedEventStart* events to make our measurements as they are independent of problems related to network instabilities and congestion. Because of server unavailability and certificate problems encountered during our crawl, we miss loading times for 43 sites. The results can be found in Table 4.4. In general, load times are extremely close between a vanilla Firefox and FPRandom. Mean times indicate a slightly better performance for Firefox. Yet, in both cases, the standard deviation is very high, meaning that the collected loading times are very dispersed between 0 and 5s. These numbers demonstrate that the modifications introduced in FPRandom do not have a visible impact on the user experience for day-to-day browsing. Moreover, we can

also say that the amount of site breakage is kept to a minimum as only a single script provided us with an error due to our enumeration modification. The modifications on both the Canvas and AudioContext API had no impact on site breakage.

Table 4.4: Web crawl results

	Times collected	Min (ms)	Max (ms)	Mean (ms)	SD (ms)
Firefox	957	10	64728	1602	3712
FPRandom	958	9	55852	1823	3935

User study

An important aspect of FPRandom is that it modifies multimedia elements that can be seen or heard by the user. To make sure that the modified subroutines do not degrade substantially the user experience at the cost of better privacy, we ran a preliminary user study in February 2017. Its goal was to compare multimedia elements as rendered by a normal version of Firefox 54 with modified ones rendered by FPRandom. The study was divided into two phases: the first was focused on the modifications made to canvas elements while the second investigated the impact on the AudioContext API. The link to our survey was communicated through regular channels like social networks and team mailing lists. We received an answer from 20 participants and the results are as follows:

- Half of them noticed a color difference between the original canvas rendering and the 10 modified ones, the other half did not.
- 70% said that some fonts made the strings harder to read and only one person said that it was significantly harder for all of the modified renderings.
- For the AudioContext API, only 25% detected a difference between the original track and the 3 modified ones.
- For people who heard a difference, they all qualified the difference with words like “tiny” or “small”.

These results give us confidence in the direction we took with our approach but we plan on conducting a more thorough study to pinpoint more precisely avenues for improvement with more participants. Still, we will investigate how we can exclude some exotic fonts as they can make strings in canvas renderings harder to read for users.

4.3.4 Discussion and further perspectives

Mitigation properties

FPRandom fulfills some of the properties defined in Section 4.1:

- FPRandom does not break browsing. As a modified Firefox browser, it behaves exactly like a native browser and our web crawl showed that the amount of site breakage is kept to a minimum.
- FPRandom is completely automatic. The tool can easily be configured and the changes to the targeted attributes are performed in the background without requiring user interaction.
- FPRandom breaks the linkability of the targeted attributes between sessions. By making some attributes unstable, they cannot be used by trackers to perform device identification.
- It is difficult to detect FPRandom without knowing the randomization strategies. As the modifications performed on the targeted attributes are very subtle to limit the impact on the user experience, it is also complicated for trackers to identify them.

Extending our approach to other vectors

With FPRandom, we aim at breaking the stability of browser fingerprints over time to improve users' privacy. By identifying APIs with restrictive implementation with respect to the JavaScript specification, we introduce randomness to produce slight variations at each execution of a fingerprinting script. We also generate noise inside HTML multimedia elements to alter their rendering without deteriorating user's perception. The approach presented in this work can be generalized to more fingerprinting vectors. For example, there exists other parts in the ECMAScript specification that leave the exact details of the implementation in the hands of developers. Routines from the *Math* object can be used to unveil information about the device and its browser [186]. If we take a look at Section 20.2.2 of the official JavaScript specification [74], it is written that "the choice of algorithms is left to the implementation" and that the behaviour of mathematical functions "is not precisely specified". This means that the actual libraries used for these functions could be diversified to prevent the unmasking of the operating system. In the end, the main challenge that remains here is to perform an exhaustive search to identify and anticipate future fingerprinting mechanisms. By locating key functions that could reveal device-specific information, we could preemptively introduce randomness to reinforce users' privacy on the web.

4.4 Conclusion

In this chapter, we detail our own defence solutions to mitigate unwanted tracking via browser fingerprinting. With Blink, we answer the problem of complete coverage found in the literature but our solution also faces some usability problems on its own. With FPRandom, we explore a new avenue for randomization but it only renders a subset of attributes unstable for tracking. In the end, there is no ultimate solution to defend against fingerprinting as the balance between privacy and usability is always tricky. Changing too many attributes in a fingerprint and the user cannot browse the web anymore. Changing too few attributes in a fingerprint and the user is still prone to tracking. It remains to be seen if the approaches explained in this thesis are sustainable as we all saw the challenges related to fooling fingerprinting systems. With the incredible pace of browser updates and new APIs, solutions that can work now can lose their utility in a matter of months.

In this fast evolving landscape of the web, there is one certainty though. Browser vendors and standard organizations have the power to change the present and future of browser fingerprinting. Acting directly at the level of web browsers and web standards is one way to fix the privacy problems raised by Internet users. Can these actors design a usable but privacy aware fingerprinting? As we saw in some of the scenarios developed in Section 3.5, the answer is a simple yes. Some modifications can be made inside web browsers to improve online privacy without breaking the way the web works. However, the important question is: will they?

Chapter 5

Augmenting authentication

Passwords are the go-to solution when it comes to authentication due to their apparent simplicity for both users and developers. However, the risks and threats on the Internet have never been higher. Every year, numerous data breaches show that logins and passwords are frequently stolen, leading to unauthorized access to accounts on the Internet. Lists of weak passwords are published showing that many users are putting themselves at risk with short or easily guessable passwords [195]. Even a big company like Yahoo recently reported account breaches from data theft [204, 205] and cookie forgery [206]. In that landscape, there is a need for more secure ways to authenticate users on the web.

The aim of this chapter is to investigate the opportunity of combining the high entropy of a canvas test with the capacity of the Canvas API to generate tests on demand in order to augment authentication. In Section 5.1, we detail the challenges behind using fingerprinting for security. As the information contained in a fingerprint can easily be manipulated, there are obvious pitfalls to circumvent to be able to use fingerprinting in real-world scenarios. We then explain why canvas fingerprinting is an ideal candidate for authentication as its dynamic nature is the key to fight off common attacks. Section 5.2 describes the preliminary analyses we performed to find the ideal tests for our system. Section 5.3 explains the inner workings of our mechanism with a description of the authentication function and the associated protocol. Section 5.4 provides a security analysis of our scheme by studying its resilience to several attacks while Section 5.5 concludes this chapter.

5.1 Using browser fingerprinting for authentication

5.1.1 Challenges

As a browser fingerprint is a direct reflection of a device and its environment, it should be the perfect conduit to augment authentication on the web. At the time of login, a server can collect very specific attributes and verify that the device belongs to the user (i.e. that the device has been seen in the past). This approach makes sense in theory as nobody changes their devices daily. However, there are three major problems that prevent fingerprinting from being widely used in practice for authentication.

- **Every attribute collected by a fingerprinting script can be manipulated by the browser.** With a simple browser extension, HTTP headers and JavaScript objects can be changed to tell any story. This opens the door to impersonation attacks as attackers can modify their own browser fingerprints to look like the ones from their victims. Through phishing, an attacker can very easily collect a complete fingerprint. As there is no way for the server to guarantee the integrity or the authenticity of the sent information, replay or pre-play attacks are then possible.

- **Fingerprinting scripts always rely on the same tests.** The literature reports on the properties and functions that are known to give away device-specific information. The problem is that all these attributes are well-known to developers but also to attackers. In order

to fool an authentication system, attackers just have to collect a fingerprint with all known attributes so that they will have all the information necessary to bypass the security put in place.

- **Browser fingerprints evolve through time.** Modern devices are constantly updated to fix security vulnerabilities, improve performance and provide new features. The consequence of this behaviour is that the exhibited fingerprint changes with time. In order to use fingerprints for authentication, a server must take into account these changes as they are an integral part of the life of device fingerprints. However, a problem arises when comparing two fingerprints. If there is a difference between two fingerprints, does it come from a genuine update of the system or does it come from a completely different device? In the first case, the system would allow the right device to connect to the user's account. But in the second case, it would allow an attacker with a very similar fingerprint to bypass the authentication scheme. Mastering the evolution of browser fingerprints is a challenging research area as data is needed to understand what changes are considered legitimate from those that are not.

All in all, using browser fingerprinting for authentication presents many obstacles. Current tests are predictable and attackers can modify their fingerprints to impersonate other devices. In order to overcome these obstacles, we need to turn to the dynamic side of browser fingerprinting where we can run a different test suite at every connection.

5.1.2 Exploiting the dynamic nature of canvas fingerprinting

Key insight

The key insight of this chapter is that canvas fingerprinting [2,13] can be used for challenge/response-based authentication. By opposition to most browser fingerprinting techniques that query an API and collect simple and deterministic browser-populated values, canvas fingerprinting can be extremely dynamic and unpredictable. Every time the user logs in, the browser can be asked to paint a canvas element in a very specific way following a newly received challenge. Strings with their own size, rotation, color and shadow can be rendered in a canvas element, exposing differences between devices. Since a canvas rendering is computed dynamically through a process that relies on both the software and hardware of the device, it is much harder to spoof than static attributes, making it a strong candidate for authentication.

Integrating a multi-factor authentication scheme

As many websites are turning to two-factor authentication (2FA) and even multi-factor authentication (MFA) [190], we designed a scheme that acts as an additional security layer in a MFA system. At each login, we ask the browser to paint two canvas elements in a very specific way: one that is used to verify the current connection and the other that will be used to verify the next connection. Because canvas fingerprinting is a stable process, the system expects that the received response will match with pixel-precision the response generated during the previous connection. If there is match, the device is authorized to continue. If any irregularities are detected, the canvas test fails and the device will be denied access.

5.2 Tuning canvas fingerprinting for authentication

Canvas fingerprinting will be suitable for authentication if it can fulfill the following properties: it is possible to generate challenges that are rendered differently on different devices (canvas tests exhibit diversity between devices) and canvas rendering is a deterministic function (the identity of a device does not change over time). In this section, we present the preliminary experiment that we ran to demonstrate that canvas fingerprinting can satisfy those properties.

5.2.1 Exploring a relevant parameter space

The canvas HTML element acts a drawing board that offers powerful tools to give life to any shapes, forms or strings that one can imagine. We use the capabilities and the features of the Canvas API to their full potential to expose as much diversity as possible between devices (i.e. what types of renderings depend the most on the device's configuration).

We perform this study in three phases. The goal of the first phase is to determine the impact of each drawing method of the canvas API for identification purposes: by drawing a wide range of shapes and by rendering strings in various ways, we are able to identify what type of drawing has the most impact on the diversity of renderings. Then, in the second phase, we explore in greater details the most promising methods by fine-tuning the parameters of the selected methods. Finally, the third phase explores potential enhancements to further increase the overall diversity. This study was performed on the AmIUnique.org website between January 2016 and January 2017. The participants were fully aware of the experiment and had to click on a button so that the tests were performed in their browser. You can find below the highlights of this study. The complete list of results for the three phases can be found in Appendices E.1, E.2 and E.3.

Entropy

We use the Shannon entropy to analyse the results of our study and know the underlying probability distribution of each of our performed tests. Let H be the entropy, X a discrete random variable with possible values $\{x_1, ..., x_n\}$ and $P(X)$ a probability mass function. The entropy's formula is as follows:

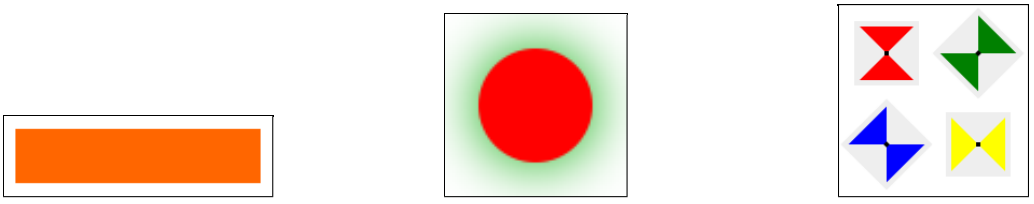
$$H(X) = -\sum_i P(x_i) \log_b P(x_i)$$

We use $b = 2$ to have the entropy in bits. One bit of entropy reduces by half the probability of an event occurring or, in our case, of a variable taking a specific value. For our study, the higher the entropy is, the better it is for our mechanism since it means that more diversity is exhibited between devices.

Phase 1

The first phase ran between January and March 2016 and 15,010 series of 15 canvas renderings were collected during that period. The 15 tests ranged from drawing simple shapes like a single rectangle to displaying more complex forms with a rich gradient.

Tests with low entropy Without any surprises, tests with simple shapes like the ones in Figure 5.1 do not provide enough ground for distinguishing devices. By specifying a shape with a uniform color, the majority of collected results are identical. Rendering curved forms helps increasing the overall entropy of a test since browsers must apply anti-aliasing algorithms to soften the edges of ellipses but, overall, all these images have an entropy that is much lower than what we can observe in more complex tests. In order to benefit from a more powerful solution, we have to turn to other features of the canvas API that rely much more on the system.



(a) One rectangle (Entropy: 2.32 bits)

(b) One circle with a shadow (Entropy: 4.89 bits)

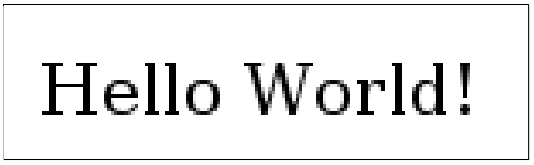
(c) Four squares (Entropy: 4.66 bits)

Figure 5.1: Examples of tests with low entropy

Tests with high entropy Rendering a string in a canvas element relies on the fonts that are installed on the system. A script can request any fonts it desires and if the system does not have it, it will use what is called a fallback font. Depending on the operating system, this fallback font differs between devices increasing the pool of distinct canvas renderings. Thanks to this fallback mechanism, tests that render strings come naturally at the forefront of our first phase. By asking for a font that does not exist and forcing the browser to use its fallback font, the diversity exposed in rendering a string is much greater than the one observed from rendering simple shapes. The wealth of available fonts on modern systems make the use of strings a prime candidate for our authentication mechanism.

Moreover, as noted by our study in Chapter 3, using emojis helps distinguishing devices because their actual representation differs between systems. An emoji is a small ideogram representing an emotion (one is present at the end of Figure 5.2c). In our series of tests, all strings rendered with an emoji have a higher entropy than those without.

Finally, the use of color gradients has a definite impact on the entropy. The natural transition between colors accentuates differences between devices than with just a single color but we leave a broader analysis of their impact for the second phase of our study.



(a) Simple string (Entropy: 6.09 bits)



(b) Simple string with a stroke and a gradient (Entropy: 6.52 bits)



(c) String with a pangram and an emoji (Entropy: 6.99 bits)

Figure 5.2: Examples of tests with high entropy

Phase 2



Figure 5.3: Basis of the tests of Phase 2 (Entropy: 7.69 bits)

The second phase ran between March and June 2016 and 21,231 series of 15 canvas renderings were collected during that period. It started directly after the first one concluded. Tests for the second phase were designed to investigate the three major distinctive components found in the first phase which are strings, color gradients and emojis. The base rendering for all the tests from this phase is represented in Figure 5.3: one string using a fallback font with all the letters of the alphabet, an emoji and a radial gradient composed of three colors.

Strings We investigated several changes to the parameters relating to the rendered string. As indicated by our test in Figure 5.4, adding a stroke actually lowers the overall diversity since the resulting entropy is lower. However, increasing the size of the used font proves to be the most impactful change since the test where we doubled the font size has the highest entropy value of our whole study (8.32 bits).



Figure 5.4: Zoom on the base rendering with text stroke (Entropy: 7.15 bits)

Gradients Modifying the gradient parameters of the string has no real impact on the overall entropy. Changing the type of the gradient from radial to linear or changing the number of colors produces a variation smaller than 0.1 bit.

Emojis Changing emojis has no impact on the entropy. The test with a different emoji has the exact same entropy value than the one from the base rendering (7.69 bits). This behaviour is expected since all the emojis of a system are most of the time provided by a single font of the system. All devices who share an identical canvas rendering with an emoji will also share an identical rendering with a different emoji since these devices share the same emoji font.

In the end, this second phase informs us that bigger renderings lead to better results but other parameters can be changed without having a real impact on the overall diversity.

Phase 3

The third and final phase ran between November 2016 and January 2017 and 23,238 series of 10 canvas renderings were collected during that period. The focus of this phase was to find further enhancements to increase the overall diversity of renderings by testing special types of curves and assessing the impact of shadows. The base rendering for this phase is the same as the previous phase (represented in Figure 5.3).



Figure 5.5: New effects from Phase 3 (Entropy: 8.11 bits)

Bezier curves The Canvas API offers the possibility to draw cubic and quadratic Bezier curves. Figure 5.5 presents one quadratic curve (the longest one) and a cubic one. These curves have a marginal impact on rendering (increase of around 0.07 bits in entropy).

Shadows A shadow can be applied to any element drawn in a canvas. Its size and its color can be changed by modifying the *shadowBlur* and the *shadowColor* properties. Shadows allow an increase in entropy of 0.16 bits.

Combining both features The interesting conclusion of this phase is that the combination of both presented features has the advantage of generating more graphical artefacts than just the sum of its parts. The entropy increases by 0.43 bits from the base rendering which is a good indication that curves and shadows are relevant additions to expose even more diversity between devices.

5.2.2 Understanding canvas stability

In parallel with our study on canvas diversity, we performed a study on the stability of canvas rendering. Indeed, if a device produces many different results in the span of a day or a week for the exact same set of instructions, our system must take that behaviour into account when asserting the validity of a response. Thanks to the extension described in Section 3.6.1, we can follow any evolution of a canvas rendering on the same device and understand the changes. If after several months, the number of canvas changes is very small or null, this would mean that the canvas API is suitable for our authentication mechanism.

Results Table 5.1 displays the main results of this study. Each column is a timespan for which we observed certain devices (difference between the first and the last fingerprint record for a device). This way, we can include every device that participated in our experiment even for a really short period of time. The mean and standard values for each category were computed without taking the outliers into account. A device is considered as an outlier if more than 8 canvas changes were observed during the given time period. They represent less than 5% of our dataset.

Table 5.1: Results on the number of canvas changes for different periods of time

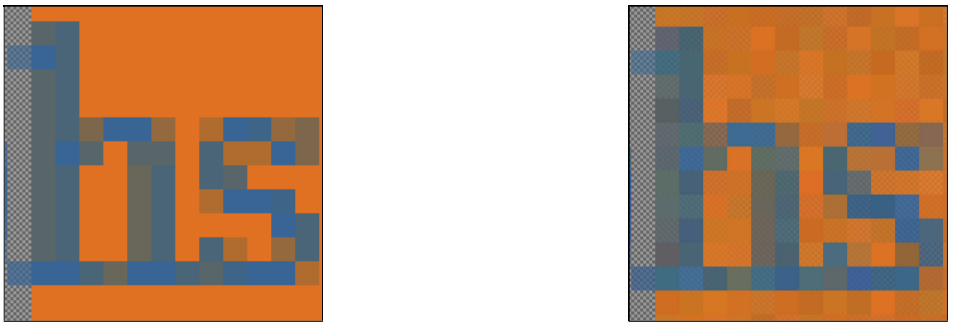
Number of days	30-60 days	60-90 days	90-120 days	120-150 days	150-180 days	180-210 days	210-240 days	240-270 days	270-300 days	300-330 days	330-360 days	360-390 days	Total
Number of devices	451	242	236	156	106	107	84	68	47	42	40	27	1606
Number of outliers (> 8 canvas changes)	15	6	8	10	4	3	4	4	5	1	4	4	68
Mean number of canvas changes	1.25	1.47	1.61	2.03	1.81	2.19	2.37	2.45	2.57	2.34	3.05	3.13	
Standard deviation of canvas changes	0.82	0.98	1.03	1.54	0.98	1.34	1.54	1.53	1.41	1.31	1.96	1.85	

The first observation is that in the span of several weeks and even months, the mean number of canvas changes is really low. It is around 2 for the first six months and, as time goes by, it is slightly increasing to be above 3 for the longest periods of our dataset. Even for the 27 devices that used the extension through the span of a whole year, half of them present 3 or less canvas changes. This means that a canvas rendering can be even more stable than the browser's user-agent which changes every 6 weeks when a new browser version is released. Moreover, the really small values for the standard deviation also prove that the majority of canvas changes are taking place in a small range between 1 and 5. These numbers comfort us in our choice for our authentication mechanism since the collected data clearly show that canvas fingerprinting can be stable through time. The canvas fingerprinting process is deterministic enough to be usable as a means of verification.

The second observation is that it is uncommon to find a device where the canvas rendering has not changed on a period of several months. This can be explained by "natural changes" of the browsing environment. For example, we noticed that some devices on Windows had an automatic canvas change when they switched from Firefox 49 to 50 because Mozilla added built-in support of emojis directly in Firefox with the bundled EmojiOne font [76]. We say these changes are natural in the sense that they are not provoked by installing a special protection against fingerprinting but they are caused by a "natural" update of the system and its components (e.g., a browser update or a change of graphics driver). In order to deal with these changes, our canvas mechanism relies on an additional verification from the authentication scheme to confirm the device identity as described in Section 5.3.2.

Outliers When we started analysing the data, we noticed some unusually high number of changes. Several devices reported different renderings every day and even one device reported 529 changes in less than 60 days. Due to the nature of the research subject, the extension attracts a biased

population interested in the subject. After investigating these high numbers of changes, we found out that some users deliberately modified their browser's configuration to produce variations in their own canvas renderings. Others have installed canvas poisoners that add a unique and persistent noise to the generated images. Color values of pixels are changed to produce a unique rendering every time the test is run. Poisoning a canvas element can be done via a browser extension like the Canvas Defender extension [47] or it can directly be a built-in feature. Pale Moon is the first browser to include such a feature in 2015 [138]. Figure 5.6 illustrates the modifications done by a canvas poisoner.



(a) Without a poisoner (b) With a poisoner

Figure 5.6: Impact of a canvas poisoner on a rendering

On Figure 5.6a without the canvas poisoner, the surface of the orange background is uniform and all pixels have the exact same color value. On Figure 5.6b, the canvas poisoner modifies the RGB values and the Alpha channel of the defined pixels and it creates a unique rendering at every execution.

5.3 Canvas authentication mechanism

We present in this section the core of our authentication mechanism and we describe the protocol associated with it. We also detail where it can fit within an already-existing multi-factor authentication scheme to augment it and reinforce its security.

5.3.1 Challenge-response protocol

The aim of this protocol is to define a series of exchanges so that a prover (client) can authenticate himself to the verifier (server). Our core mechanism relies on the comparison of images generated through the canvas browser API. During one connection to the verifier, the prover uses our authentication function to generate a very specific image. Our mechanism will then verify in the next connection that the device can generate the exact same image.

For our comparison, we only look to see if two generated images are identical to one another. Indeed, if two images differ even by a very low number of pixels, it could legitimately be caused by an update of the prover's device or it could have been generated by a completely different device altogether. For this specific reason and to prevent making false assumptions by incorrectly identifying a device, we do not compute any similarity score between two images and we do not use specific thresholds. Moreover, we showed in Section 5.2.2 that a generated image presents a strong stability through time so we can perform pixel-precise comparisons without worrying about constant changes. For devices using a fingerprinting protection like a canvas poisoner, we expect users to whitelist websites that implement our canvas mechanism to avoid changes at every connection. The Brave browser has already adopted a similar approach as its "fingerprinting protection" mode, which blocks several APIs and notably the Canvas one [43], is lifted on websites that are present in a "Saved Site Exceptions" list.

The challenge-response protocol underlying our authentication mechanism is depicted in Figure 5.7.

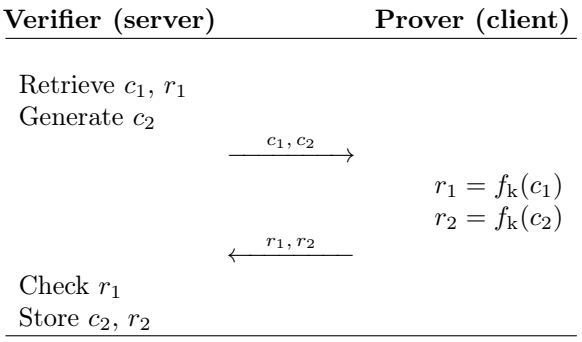


Figure 5.7: Overview of the authentication protocol

Step n°1 The verifier sends two challenges c_1 and c_2 to the prover when the connection is initiated:

- the challenge c_1 is one that has already been answered by the prover during the previous connection. It is used to authenticate the device for the current connection by comparing the response to the one given in the last authentication process.
- the challenge c_2 is randomly generated. The response r_2 given by the prover is stored alongside the associated challenge c_2 and they will be used to authenticate the device for the next connection.

The verifier is trusted to generate challenges in a way that they are indistinguishable from random values.

Step n°2 The prover executes the authentication function f with each challenge and sends the resulting renderings to the verifier. The secret key k represents the prover's device and its configuration. Each image is sent to the verifier in a textual representation obtained with the "getDataURL()" function of the canvas API.

Step n°3 This is the step where the verifier confirms the identity of the prover. The verifier asserts whether the given response to the first challenge matches the one of the previous connection. If the given canvas rendering is identical to the stored one (i.e. if the obtained strings are identical), we say that the response is valid and the authentication process is a success. If not, the authentication system will follow through by asking for an additional confirmation through another channel (see 5.3.2).

Assumptions on the channel A confidential channel (e.g., based on HTTPS) must be used between the verifier and the prover to avoid a trivial replay attack of a genuine authentication exchange. Moreover, there is a bootstrap phase for our mechanism because the verification process cannot be performed during the very first connection to our system where only a single challenge is sent. As defined in the adversary model in Section 5.4.1, we assume that prover and verifier share an authenticated channel in the very first connection.

5.3.2 Integration in a MFA scheme

As seen in Section 5.2.2, a single canvas rendering from the same computer evolves through time and it can take weeks or months before a difference can be observed. Because of these changes, our system cannot stand on its own and substitute a complete MFA scheme. Here, the main goal of our canvas mechanism is to strengthen the security provided by a multi-factor authentication scheme and works alongside it. It can be integrated in traditional multi-layered schemes like many 2FA ones or it can be used as an additional verification technique for authentication protocols like OpenID Connect [132] or Facebook Login [81] that provides end user authentication within an OAuth 2.0

framework [128]. Figure 5.8 gives an overview of how our system can fit within an existing MFA scheme. A more complete example with greater details can be found in Appendix E.4.

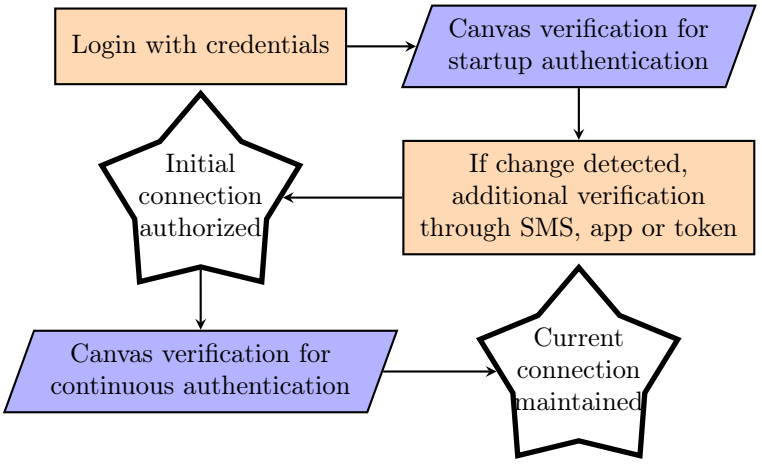


Figure 5.8: Overview of the integration of the canvas mechanism in a multi-factor authentication scheme

Login verification First, the user must provide his credentials. For a lot of websites, it represents the combination of a username, a password and the presence of a cookie in the browser. After this phase is successfully completed, our canvas mechanism takes over to perform the necessary verification. If the responses from the canvas tests are valid, the user is authorized to continue. If the information sent by the browser is labelled as incorrect, an additional means of verification will kick in like the use of a one-time password (OTP) via an SMS or an app. As shown in our stability study, it can take many weeks for a single canvas rendering to undergo a change due to a "natural" evolution of the user's device. The expected impact on the end-user experience is minimal as going through an additional verification step every 3 months is acceptable for the user.

Continuous authentication The particularity of our system is that it can be used beyond the simple login process to protect against session hijacking without adding another usability burden for the client. We can use our mechanism to continuously authenticate the client because canvas renderings do not present changes in the span of a session and the authentication process is completely transparent to the user. Every few requests, we can recheck with our system the identity of the client's device and it has the benefit of hardening the work of an attacker wanting to defeat our authentication scheme.

Managing separate devices It should be noted that each device has to be registered and authorized separately. This behaviour is exactly the same as many websites which currently enforce 2FA. A simple analysis of the user-agent with other finer-grained attributes are sufficient to distinguish devices belonging to the same user. During the registration process of a new device, our canvas mechanism is bootstrapped so that it can be used in subsequent connections for verification. Moreover, we have to point out that we keep a separate series of canvas tests for each device since they have their own software and hardware layers.

5.3.3 Authentication function

Section 5.2.1 showed that large canvas renderings with strings and color gradients are the prime candidates to exhibit diversity between devices. Shadows and curves are also small enhancements that can be utilized to further increase the gap between devices. Building on these results, we detail how we use the canvas API to generate unique challenges (i.e. unique test parameters) at each connection.



Figure 5.9: Example of a canvas test

Figure 5.9 shows an example generated by our authentication function f: random strings and curves with randomly-generated gradients and shadows. The function is not secret and is identical for everyone. The secret that the authentication relies on is the device configuration. The set of parameters that we randomize are defined below.

Content of strings Each font has its own repertoire of supported characters called glyphs. We use basic alpha-numerical glyphs that are guaranteed to be supported by most fonts. The generated strings in our challenges are 10 characters long with any combination of glyphs.

Moreover, the order of letters in a string can be important and not all fonts behave the same way. Glyphs in monospaced fonts share the exact same amount of horizontal space while glyphs in proportional fonts can have different widths. Each proportional font contains a kerning table that defines the space value between specific pairs of characters. OpenType fonts also support contextual kerning which defines the space between more than two consecutive glyphs.



(a) Arial (b) Courier New

Figure 5.10: Spacing comparison between fonts

Figure 5.10 illustrates the kerning mechanism. With Arial which is a proportional font, the kerning table specifies a negative space value for the pair “Ta” but nothing for the pair “aT”. For Courier New which is a monospaced or fixed-width font, each string occupies the exact same amount of horizontal space. In the end, this difference of spacing helps us increase the complexity of our test.

Size and rotation of strings Our experiments detailed in Section 5.2 show that bigger canvas renderings are better at distinguishing devices than smaller ones, because they are more precise and finer-grained. Also, as shown in Figure 5.11, rotating strings leads to pixelation and requires partial transparency to obtain smooth edges. In our case, a bigger image leads to softer but well-defined limits between pixels.



(a) Zoom on a ‘T’ character before rotation (b) Zoom on a ‘T’ character after rotation

Figure 5.11: Details of a letter ‘T’ showing pixelation after rotation

Curves While the impact of curves or rounded forms is definitely smaller than the one with strings, they can still provide an added value to our mechanism. Indeed, the shape of letters in a font can be very straight with sharp and squared corners. Relying only on these letters can prevent us from displaying rounded strokes that generate graphical artefacts from aliasing effects. To increase the complexity of our authentication function, we generate cubic and quadratic Bezier curves. A curve is defined by starting and ending points along with a variable number of intermediate ones (two for cubic curves and one for quadratic curves).

Color gradient With the introduction of CSS3, browsers now support linear and radial color gradient. A gradient can be seen as a natural progression from one color to the other and a CSS gradient element can be used on any string to change its color. An unlimited number of points can be defined in a gradient and every single one of these points can have its own color from the whole color palette. However, the main challenge is to find the right balance between the size of the string and the number of colors. Indeed, if the size of the rendered string is too small and the gradient is comprised of a thousand different colors, the rendering will not be big enough so that all transitions are visible resulting in a meaningless cluster of pixels.



Figure 5.12: Identical strings with different color gradients

Shadow A shadow is defined by its color and the strength of its blur effect. A very small strength value will cast a very thin shadow around the rendered object but a higher value will disseminate a small cloud of pixels all around it (see Figure 5.13).



Figure 5.13: Identical forms with different shadow blurs (strongest blur on the right)

Number of strings and curves All the strings and curves in a canvas test have their own styles from their color gradient to their shadow. The overall complexity of our test can be increased by generating dozens of strings and curves without impacting the performance of the authentication mechanism.

5.3.4 Implementation

We developed a prototype of our mechanism in JavaScript and we looked at both the performance and the diversity of generated challenges.

Steps and performance The first step of our prototype generates a random challenge for the prover which includes the exact content, size, position and rotation of all the strings along with specific values for shadows and curves. The average generation time on a modern computer is less than 2ms.

As a second step, the prover executes the authentication function with the received challenge. Depending on the browser used, this step can take as much as 200ms for the most complex challenges. The average rendering time for Chrome users is about 50ms while it is around 100ms for Firefox users.

Finally, since we collect a textual representation of an image, complex challenges can result in really long strings with more than 200,000 characters. To prevent storing these long strings in the authentication database, it is possible to use a cryptographic hash function to hash all received responses, reducing so the stored values to 128 bits. The recorded average hashing time is around 40ms.

Overall, the complete process takes less than 250ms which is very acceptable in the case it is used in an authentication scheme.

Evaluation To make sure that our mechanism is supported by most browsers, we deployed our script alongside the one used for our test described in Section 5.4.5. We generated a unique challenge for more than 1,111,000 devices and collected the associated response for each of them. We also checked that two challenges with supposedly different parameters would not produce two identical renderings (e.g., in the case that two challenges are different on paper but translate into identical forms and figures in the final image). The analysis shows that 99.9% of devices returned a canvas rendering and all collected images were unique. The 0.1% of non-unique values come from either older browsers that do not support the canvas API and returned identical strings, or browsers with extensions that blocked the rendering process and returned empty canvases. This result shows that the canvas API is supported by most devices and it gives us confidence that the challenges generated for our authentication function exhibit enough diversity.

5.4 Security analysis

In this section, we define the adversary model and analyse the security provided by our authentication system against known attacks.

5.4.1 Adversary model

The goal of the adversary is to impersonate a prover and fool the verifier into believing that the sent images come from the prover. With this goal in mind, we define the adversary model as follows:

- The adversary cannot tamper with the verifier and modify its behaviour.
- The adversary cannot eavesdrop messages exchanged between the verifier and the prover. We ensure this by setting up a confidential tunnel for every connection.
- The adversary cannot interfere with the very first exchange between the verifier and the prover. As with any password-based online authentication system, the first exchange must be authenticated by another channel, e.g., a confirmation email during the registration step.
- The adversary does not know the prover's device configuration, and neither does the verifier. By analogy with keyed cryptographic functions, the device configuration is the prover's secret key.
- The adversary knows the authentication function.
- The adversary can set up a fake verifier and query the prover with a polynomially-bounded number of chosen challenges.

5.4.2 Replay attack

Example An attacker listens to the Internet traffic of the victim and eavesdrops on a genuine connection between the server and the client. His goal is to collect the victim's responses to the canvas challenges to replay them in a future connection.

Analysis Before the protocol defined in Section 5.3.1 is executed, we assume that a secure channel is established between the verifier and the prover. This can be easily achieved, for example with TLS, which is today widely deployed on the web. Given the properties of a secure channel, a replay attack cannot occur.

5.4.3 Man-in-the-middle or relay attacks

Example An attacker creates a phishing website that masquerades as a trustworthy or legitimate one. To get access to the service, the victim will enter his or her credentials not knowing that the data is directly transmitted to the attacker. The fake server can then ask the victim's browser to compute any canvas challenges while the connection is active.

Analysis Introduced in 1976 by Conway with the Chess Grandmaster problem [56], relay attacks cannot be avoided by classical cryptographic means. As so, our mechanism is defeated by a relay attack as an attacker has a direct access to the device to collect any number of canvas renderings. However, if we start analysing the impact of our canvas mechanism as part of a whole MFA ecosystem as it is meant to be integrated, it can provide an added layer of security. Our mechanism can be used beyond the simple login process for continuous authentication as it is completely transparent to the user. Canvas challenges can be sent to the user's browser every few page requests to verify that the connection has not been tampered with without requiring the user's interaction. In our case, it hardens the work of an attacker as he must not only complete the login process but also maintain the connection to the victim to keep sending him the canvas challenges. Continuous authentication also works against attackers who hijack already authenticated sessions as they will face the same difficulties. For most websites, getting the user's credentials is enough to get access to the desired service but with our mechanism, we provide an additional barrier to protect a user's account.

5.4.4 Preplay attack

Example An attacker sets up a web page to collect ahead of time any canvas renderings he desires from the victim's device. The particularity of this attack is that it can be completely stealthily as it does not require user's interaction. Moreover, the attacker does not necessarily need to put some effort into building the perfect phishing opportunity as any controlled web page can run the desired scripts.

Analysis A preplay attack consists for an attacker to query in advance the prover with arbitrary challenges, expecting that the challenge that will be sent by the verifier will belong to the arbitrary challenges selected by the attacker. This way, an attacker has the necessary tools to correctly get through our authentication mechanism. In this section, we consequently analyze how likely and practical it is for the attacker to obtain responses and whether they allow him to perform an attack with a non-negligible probability.

Injectivity of the authentication function

For a given device, different challenges produce different canvas renderings because the authentication function is injective. Indeed, every single parameter that is described in Section 5.3.3 has an impact on a canvas rendering. Positioning a string and choosing its size will define which pixels of the canvas are not blank. Generating a gradient will modify the RGB channels of each pixel of a string. Rotating a string will use partial transparency to have a faithful result and to define precisely the limits between pixels. In the end, no two challenges will produce the same response on a single device. Even the smallest change of color that the eye cannot perceive will have an impact on the final picture.

Exhaustive collection of responses

Since the actual code of our authentication function is not secret, the attacker can set up a verifier that poses as a legitimate one and asks the victim any challenge he wants. In other terms, this

means that an attacker has a partial access to the victim's device to collect any responses he desires. Here, we estimate the number of possible challenges and the time it would take to transfer and store all responses. It should be noted that we do not consider the use of curves for this calculation.

Estimating the number of challenges:

- String content: Generated strings are composed of 10 alpha-numerical characters. We consider both lower-case and upper-case variants of standard letters. In total, we have 26 upper-case letters, 26 lower-case letters, and 10 figures. We so have 62^{10} combinations.
- Size: Bigger font sizes lead to a better distinction of devices. We fix the lower bound at 30 and the upper one at 78, which leads to 49 different font sizes.
- Rotation: Every string can be rotated by any specific value. Following tests we performed, a precision of the rotation up to the tenth digit has an impact on the canvas rendering. Smaller variations do not result in detectable changes. We consequently consider $360^o \times 10 = 3600$ different rotations.
- Gradient: The two parameters of a gradient are its colors and the position of each of these colors. The RGB color model is used and each color is encoded on 8 bits so we have 2^{24} different colors at our disposal. We use the "Math.random()" JavaScript function to give us the position of each color on the gradient line. This function returns a number between 0 and 1 and it has a 52-bit precision. Variations from the thousandth digits have seemingly little to no impact because of the limited number of pixels in our images. We only consider precision up to the hundredth digit and we limit the number of different colors in a given gradient to 100 with a lower bound at two (one color at each extremity of the gradient line). Considering two colors provide a conservative lower bound on the number of different gradients, we have $(2^{24})^2 = 2^{48}$ combinations.
- Shadow: The color and the blur of the shadow can easily be tweaked in the canvas API. The selection of the color is identical to the one described for the gradient so it provides 2^{24} possibilities and we constrain the strength of the blur between 0 and 50.
- $Total = 62^{10} \times 49 \times 3600 \times 2^{48} \times 2^{24} \times 51 \approx 2^{154}$ challenges

Taking into account practical implications, storing all responses would occupy 2.3×10^{50} bits with an average of 10 kb per response. It would take several quintilliard years on a Gigabit internet connection to transfer everything without considering possible network instabilities and congestion. The sheer size of these numbers eliminates all possibilities to conduct a successful attack following this approach.

5.4.5 Guessing or building the right response

Example An attack against a challenge-response protocol consists for an attacker to guess or build a valid response upon reception of a canvas challenge. Here, the attacker may have access to a cluster of compromised devices to ask for canvas renderings and he may also have specific techniques to build a response from previously observed renderings.

Analysis To defeat such an attack, the set of possible responses should be large enough, and the authentication function f should be non-malleable.

Blind guess

An attacker could blindly generate an image regardless of the sent challenge. This way, he can set any desired RGBA values for all the pixels of the response. Since the canvas size in our mechanism is 1900x300, the total number of pixels is 570,000. Given the alpha value is not random (it strongly depends on the image content), a conservative approach consists in not considering it in the analysis: the number of possible responses is then $2^{24 \times 570000} = 2^{13680000}$ which is far too high to consider this approach feasible.

Choosing the most popular response

With the help of a partner who is a major French telecom operator, we sent the exact same challenge to more than 1,111,000 devices on a period of 42 days between December 2016 and January 2017. Devices who visited the weather or politics page of the official website of this operator received this test and we collected the response for each of them. It should be noted that this test is different from the one described in Section 5.3.4 where we generated a unique challenge for each device. Here, we sent the same challenge to everyone.

In total, we observed 9,698 different responses, and 4,645 responses were received only once (i.e. a single device answered this response). Figure 5.14 shows the distribution of sets who share the same responses. 98.7% of them contain each less than 0.1% of the population. Only a single set is above the 5% threshold and it represents 9.9% of the population.

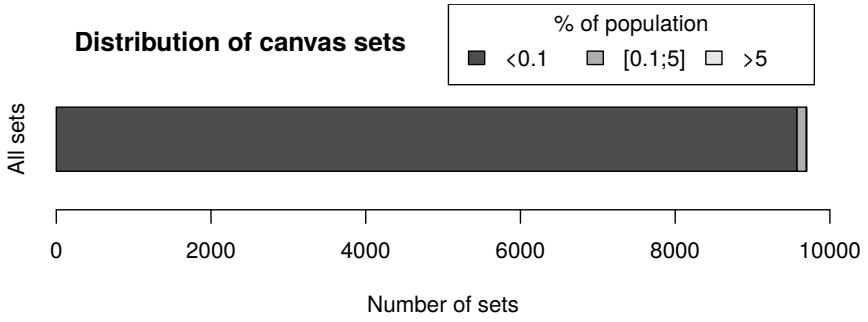


Figure 5.14: Number of sets containing x equal responses

The attacker aims to send the same response than the one the victim he seeks to impersonate would send. To do so, the attacker can request his own computer with the challenge received from the verifier. The attack succeeds if and only if the response generated by the attacker matches the victim's expected response. Calculating the success probability is related to the birthday paradox on a non-uniform distribution. Assuming p_i is the number of responses received in the set i ($1 \leq i \leq 9698$) divided by the total number of received responses, the attacker's success probability is:

$$p = \sum_{i=1}^{i=9698} p_i^2.$$

Given the numerical values collected in our experiments, we have $p < 0.01$.

Note that the attacker can increase his success probability if he knows the distribution provided in Figure 5.14. Indeed, to maximize his probability of guessing the right response, the attacker would choose the response sent by the highest number of devices. The bigger this set is, the higher his probability of fooling the system will be. Obtaining the full distribution of responses is in fact not required to maximise the probability of success: identifying the largest set is enough. To do so, the attacker can use a cluster of computers with common configurations (e.g., a botnet): upon reception of a challenge from a verifier, the attacker sends this challenge to each computer of the cluster and can expect to identify the most common canvas renderings for a given challenge. It is worth noting that this attack requires a cluster and allows the attacker to increase his success probability up to 9.9% only. This value is challenge-dependent but it seems to be rather stable.

Forging canvas renderings

Instead of guessing the response, an attacker can try to recreate a genuine response from observed ones. Although the authentication function is malleable to some extent, our experiments show that its malleability is quite limited. In particular, we consider the following actions possible: resizing glyphs, changing the place of glyphs in a string, rotating glyphs, and applying a custom gradient. We saw in Section 5.3.3 that the order of glyphs is important in proportional fonts because of kerning. If an attacker were able to learn all combinations of two letters ($52^2 = 2704$), he would not need to enumerate all the strings of 10 characters so the difficulty of knowing all

responses would be lowered. On the other hand, changing font, size, and rotating strings increase the difficulty of the attack. These operations generate distortions and create artefacts like aliasing or blurring. Interpolation between pixels must be perfect so that a rendering is faithful to an original one. An attacker will need deep knowledge in image transformation to find the right algorithm that answers the requirements imposed by our pixel-precise image comparison. Moreover, with the use of gradients and shadows, the complexity of our test is raised to another level since it becomes even harder to find the right balance between colors and transparency to achieve the perfect transformation.

Defining precisely what is possible through image modification is still an open question. We consider though that the number of involved parameters makes the attack hardly achievable even with the help of strong image transformation tools. Although simple cases can occur, e.g., a non-rotated string with the same letter 10 times and a gradient with the same color on both ends, the probability that such a weak challenge c (i.e. $f_k(c)$ does not depend on k) is randomly picked is negligible. In the end, forging canvas renderings does not seem to be a relevant attack because guessing the response is an easier attack, with a higher success probability.

5.4.6 Protection against configuration recovery

Example By getting the browser's fingerprint of the victim from a controlled web page, the attacker can try to rebuild the same configuration or even buy the same device to start from a configuration that is as close as possible to the real one.

Analysis The last attack considered in this security analysis is the configuration recovery, which is somehow equivalent to a key-recovery attack in a classical cryptographic scheme. The full knowledge of the configuration of a prover is indeed enough and sufficient to answer correctly to any challenge. Contrarily to classical cryptography, though, the key is not a 128-bit binary secret but the full hardware and software configuration of the verifier. Partial knowledge about the configuration can be obtained by the adversary using a browser fingerprinting script. This mechanism indeed provides the attacker with information on the targeted victim, e.g., the browser model, the operating system, and the GPU model. A key issue consists in evaluating how much the device configuration leaks when the prover faces a browser fingerprinting attack. It is in particular important to evaluate whether a browser fingerprint provides the attacker with enough information to build valid responses. It is worth noting that current fingerprinting techniques do not reveal the full device's configuration, e.g., they cannot catch the device's driver, kernel, and BIOS versions, to name but a few.

Our analysis considers the set of fingerprints used in the previous section, and we divided our dataset into two categories: desktops and mobile devices. This distinction is important because desktops are highly customizable whereas smartphones are highly standardized and present a lot of similarity across models and brands. 93.3% of these 1,111,819 fingerprints come from desktop devices and 6.5% from mobile ones. Less than 0.2% are discarded because they either come from bots or are devices that could not be identified (their user-agents did not give enough information). In order to determine the advantage a user can get, we regrouped fingerprints that were identical with each other and we looked at the generated responses. The collected fingerprints were composed of the following attributes: the HTTP user agent header, the HTTP language header, the platform, the CPU class, the WebGL renderer (GPU of the device), and the width, height and color depth of the screen. Inside the same group, if the canvas renderings are different from each other, this means that the fingerprints do not capture the totality of the prover's configuration (i.e. the key is partial). An attacker would then not be able to recreate faithfully the victim's renderings even if he had the same fingerprint.

Figure 5.15 shows the distribution for desktop and mobile devices. For each category, the distribution is divided into three: groups with a single rendering (i.e. all the devices in the group have generated the exact same image), groups with 1 to 5 renderings, groups with more than 5 different renderings. To be consistent with the chosen scale, we only kept groups that contained at least six fingerprints.

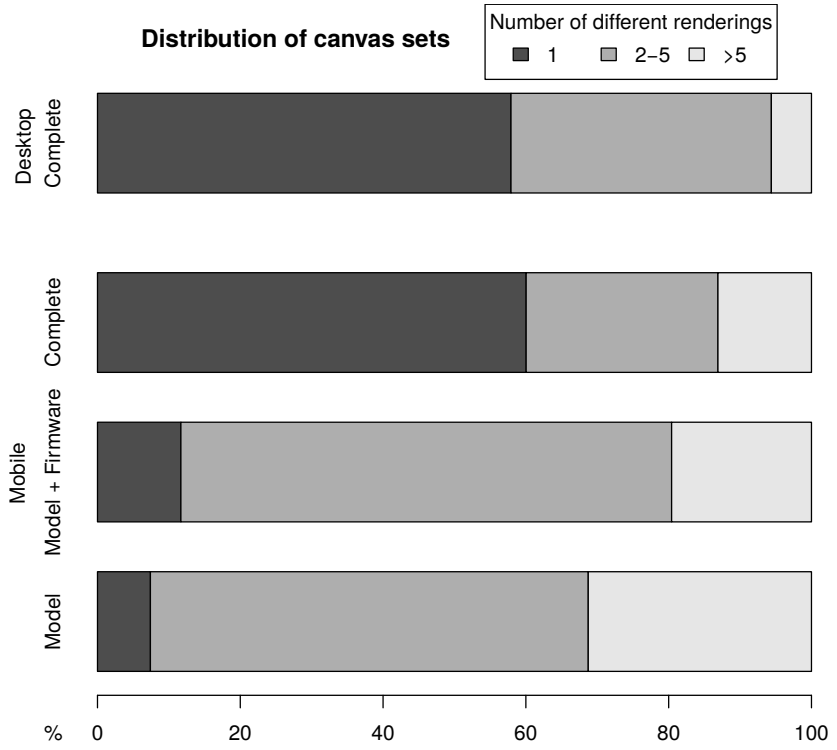


Figure 5.15: Distribution of canvas renderings for groups with identical fingerprints

Desktop With the heterogeneity of desktop configurations, we perform the analysis by keeping all the fingerprint attributes. About 57.9% of groups have devices that share the same canvas rendering. This number is pretty high but it does not give an attacker full confidence that he will produce the right rendering even if he has the same fingerprint as his victim.

Mobile On mobile devices, it can be much easier to identify the type and the model of the device. As detailed in [2], "some smartphones running Android give the exact model and firmware version of their phone" via the user-agent. Since phones with the same model have supposedly the same specifications, it is important to verify if all of them present identical renderings. In particular, one may wish to verify whether the version of the Android firmware has an impact on the canvas painting process. To find the answer, we decided to adjust the information contained in a fingerprint and created three different categories:

- Complete: We took the complete fingerprints like we did for the desktop analysis.
- Model and firmware: We extracted from the user agent the model and the firmware of the device.
- Model: We limited ourselves to the phone model only.

First, if we consider all the collected attributes, the percentage of groups with a unique rendering is 60%. This value is higher than what is observed with desktops, which was expected since mobile devices are a lot less customizable. Then, if we limit ourselves to the model and its firmware, the groups with a unique rendering drops to 11.7%. This significant drop can be explained by the fact that software customization has an impact on the canvas rendering. Notably, there are many different apps and browsers available that can influence the generation of the response. Finally, if we only identify the model of the phone, it proves to be insufficient for an attacker since the percentage drops to a meagre 7.4%. This means that buying the same smartphone as the victim still requires some work to be able to faithfully replicate a canvas rendering. It is really surprising to see that there can be a lot of diversity even when considering the exact same phone with the

exact same firmware. These numbers not only prove that our mechanism is somehow resilient to configuration recovery on desktops but also on smartphones.

In the end, even if the knowledge of a browser fingerprint makes easier a configuration-recovery attack, the problem can be mitigated if the prover does not fully reveal his fingerprint, or lies about it. It is also worth noting that this attack requires extra work as the attacker has to set up a computer whose fingerprint is the same as the one of his victim, or manage a cluster of computers whose at least one possesses the expected fingerprint.

5.5 Conclusion

Using browser fingerprinting for authentication is challenging. Attributes in a fingerprint can be captured, manipulated and replayed, opening the doors to many different attacks on the web. Most of them are also predictable and the method to collect them are widely-known by attackers. In this chapter, we show how canvas fingerprinting can be used to reinforce security on the web despite these problems. Notably, we show that it can act as an additional layer in a multi-factor authentication scheme. Our authentication mechanism is completely transparent and frictionless for the user. It leverages differences in software and hardware between devices by asking a browser to paint a canvas element in a unique way. We ask browsers to display a series of custom strings with their own size, rotation, shadow and colors. Depending on the browser and its version, the operating system, the GPU, and the list of fonts, the generated canvas rendering differs between devices. The key concept of our proposed mechanism is that we exploit the dynamic nature of canvas fingerprinting for authentication because new parameters can be generated at every connection. This way, attackers cannot bypass our system as they will not know the answer beforehand.

By harnessing the power and possibilities offered by modern browsers, canvas fingerprinting can be used to harden the work of an attacker. It cannot replace a classical cryptographic authentication protocol but it can provide for free an added layer of security in a multi-factor authentication scheme. While our system focuses on using the Canvas API, the detailed approach can be extended to other dynamic techniques, namely WebGL and Web Audio fingerprinting. However, a complete analysis must be performed on them to outline their exact limits when used for authentication.

Chapter 6

Conclusion

6.1 The big picture

The web is a beautiful platform and browsers gives us our entry point into it. With the introduction of HTML5 and CSS3, the web has become richer and more dynamic than ever and it has now the foundations to support an incredible ecosystem of diverse devices from laptops to smartphones to PCs and tablets. The diversity that is part of the modern web opened the door to device fingerprinting, a simple identification technique that can be used to collect a vast list of device characteristics on several layers of the system. As its foundations are rooted into the origin of the web, browser fingerprinting cannot be fixed with a simple patch. Clients and servers have been sharing device-specific information since the beginning to improve user experience as a whole.

As we saw in this thesis, the main concept behind browser fingerprinting is straight-forward: collecting device-specific information for purposes like identification or improved security. However, when this concept is implemented, its exact contours are constantly changing as its mechanisms are entirely defined by current web browser technologies. Each new browser version that adds, modifies or even removes an API has a direct impact on the domain. Each new draft that is written by the W3C introduces new capabilities. Browser vendors and standard organizations are continually shaping the future of browser fingerprinting as they discuss about what is next for the web and what will run in tomorrow's browsers.

The fascinating aspect about browser fingerprinting is that it is a technique at a crossroad between companies, academic research groups, law makers and privacy advocates. As it got out of the research lab, it has a concrete impact on the web as it is now used in real-world scenarios. For business companies, browser fingerprinting represents an alternative to current methods of tracking and identification at a time where the ad landscape is undergoing tremendous changes with the rise of ad blockers. For research groups, browser fingerprinting brought unexpected questions about the privacy status of current and future web APIs. Especially, the work done by researchers on the Battery API exposed the possible privacy problems that could be hiding in current browsers. It also highlighted the importance of having third parties review the drafts of the W3C before being implemented by browser vendors. For law makers, browser fingerprinting represents an additional tracking mechanism that must be regulated so that the control is given back in users' hands. However, as touched upon in this thesis, this technique is a side-effect of current browser architecture and no browser provides built-in blocking systems to disable use of fingerprinting. While the presence of cookies can be verified by checking the different browser storage mechanisms, detecting that a browser is being fingerprinted is much more complex. Regulators are already in an uphill battle before it has even begun to control uses of this technique. Finally, individuals and entities like journalists, activists, businesses or members of the military that rely on the confidentiality and privacy of their communications for their day-to-day activities must now take browser fingerprinting into account. While in the past they had to think about securing their connections through a VPN or a proxy, they must now think about what could transpire through their browsing environment as it could potentially compromise their identity.

All in all, browser fingerprinting is still a fairly new technique. Researchers and developers are

still trying to understand what is possible with it as we are coming to grasp with its intricacies. Its future is paved with uncertainty, but if the past seven years are any indication of what is to come, the road ahead is definitely full of surprises.

6.2 Contributions

In the thesis, we aimed at extending the state of the art on browser fingerprinting while exploring at the same time a new application for the domain.

6.2.1 The past, present and possible future of browser fingerprinting

With the AmIUnique.org website, we analysed 118,934 browser fingerprints collected from online visitors. The study provides a much needed update on the actual state of browser fingerprinting on the web. First, we confirm Eckersley's findings in 2010 with his Panopticlick experiment as 89.4% of collected fingerprints were unique. However, in the 6 years that separated both studies, we saw an evolution in the different attributes that compose a fingerprint. While the list of plugins and fonts were kings at the the beginning of the decade, it is not the case anymore as plugins have been deprecated in major browsers because of the security threat they pose. Newcomers like canvas fingerprinting provide very strong results as we observed an important entropy in the collected values. Then, at a time where the use of smartphones is booming, we show that mobile fingerprinting is possible but for different reasons than on desktops. In our dataset, 81% of fingerprints from mobile devices are unique. HTTP headers and HTML5 canvas fingerprinting play an essential role in identifying browsers on these devices. Finally, we simulate scenarios to assess the impact of future web evolutions. We show that certain scenarios would limit the detriment these technologies have on privacy, while preserving the current trend towards an ever more dynamic and rich web. In our study, simple changes like having generic HTTP headers or removing plugins reduce fingerprint uniqueness in desktops by a strong 36%.

6.2.2 Improving current defence mechanisms

Our approach when designing strong countermeasures against browser fingerprinting is to increase the diversity of exhibited fingerprints. By presenting constantly changing values, we break the stability of a fingerprint over time and third parties are then unable to track a device.

One of the major drawback of defence mechanisms that modify the content of fingerprints is that they provide an incomplete coverage of attributes. While one attribute tells a story, another unmodified attribute is telling a different one proving that the browser has been tampered with to provide protection. With a solution called *Blink*, we aimed at fixing this problem. *Blink* relies on a large pool of components composed of 39 plugins, 2,762 fonts, 6 browsers and 4 operating systems. When the user wants to browse the web, components are chosen randomly and they are assembled to create a short-lived browsing session. The intuition behind Blink is that the fingerprint of a session will be so different than the fingerprint of the previous one that no third parties will be able to link the browsing activities of the user. The big advantage of *Blink* is that the exposed fingerprint is genuine with no inconsistencies as the system utilizes components that truly run on the user's device. If a script tries to find values that are incompatible with each other, it will not find one as the environment exists and attributes have not been modified artificially. By generating 5,000 random browsing environments, we show that Blink is capable of assembling very dissimilar configurations and it can deceive a commercial fingerprinting script from BlueCava.

With *FPRandom*, we aimed at breaking the stability of specific attributes directly inside the browser. We looked at the JavaScript specification and we identified that some browsers functions had deterministic implementations with respect to their specification. We decided to introduce randomness in some of these functions so that they produce slight variations at each execution of a fingerprinting script. The *FPRandom* prototype targets canvas fingerprinting, AudioContext fingerprinting and the unmasking of browsers through the enumeration order of JavaScript properties. Our evaluation proves that *FPRandom* provides an effective protection by running it against

known fingerprinting tests. The performed benchmarks also show that the overhead introduced by our solution is very small, thus having a minimal impact on the user experience.

6.2.3 Augmenting authentication

Using a fingerprint for authentication poses a major challenge as it could be modified to impersonate another device or it could have been replayed from a previous connection to a victim. The key insight of our contribution is that we use canvas fingerprinting for challenge/response-based authentication. By opposition to the simple and predictable browser-populated values, canvas fingerprinting is extremely dynamic. Every time the user logs in, the browser is asked to paint a canvas element in a very specific way following a newly received challenge. Since a canvas rendering is computed dynamically through a process that relies on both the software and hardware of the device, it is much harder to spoof than static attributes, making it a strong candidate for authentication. Thanks to the help of 1,606 devices, we show that canvas fingerprinting is stable enough to be used for authentication as we report an average of only three canvas changes in the span of a whole year. With the analysis of images collected from more than 1,111,000 devices in a real-world large-scale experiment, we also show that our scheme provides great diversity and a negligible performance overhead. Finally, we provide a security analysis of our mechanism and establish that it fights off common impersonation attacks and configuration recovery attacks.

6.3 Future work

6.3.1 Detecting fingerprinting scripts

Finding fingerprinting scripts on the Internet is important for two reasons: to know how widespread this technique is and to detect new fingerprinting techniques that are not reported in the literature. In order to quantify the number of websites that are currently using fingerprinting scripts on the Internet, one needs the means to identify them. At first sight, the task may not appear too difficult as the collection process in a browser is straightforward but the reality is in fact much more complex. If a script accesses the user-agent header and the list of plugins, it could be for legitimate purposes to tailor the current web page to the user's device. But it could also be the first-step towards building a complete browser fingerprint. If a script makes hundreds of calls to the Canvas API, it may be rendering a 2D animation in the browser. But it may also probe for the list of fonts installed on the system. These simple examples illustrate that the line between a benign script and a fingerprinting one is far from being clearly defined. When crawling the web, researchers are facing a lot of challenges to classify scripts correctly as the goal of two scripts can vastly vary even if they present very similar content.

Here, we detail several signs that indicate that a script may be partaking in fingerprinting activities.

- **Accessing specific functions** In the fingerprinting literature, many functions and objects are known to return device-specific information (see Section 2.3 for more details). For example, the *navigator* object contains the user-agent and the platform. Does the script access these very specific functions and objects?
- **Collecting a large quantity of device-specific information** Even if a script access the screen resolution, this information alone is not sufficient to identify a device on the Internet. If a script queries specific APIs, how many of them are accessed? Can the collected information be used to identify a single device?
- **Performing numerous access to the same object or value** If a function is called an incredible number of times, can it be considered as a normal usage of the API? Or is the script testing different parameters to expose a certain property of the device? How can we consider a usage as normal or abnormal?
- **Storing values in a single object** Is the script storing all collected values in the same object? From a design perspective, having all the values in the same object means that they probably share a similar purpose.

- **Hashing values** Scripts can hash very long strings to ease processing, transfer or server-side storage. Is the script hashing any value, especially ones that come from known fingerprinting functions?
- **Creating an ID** Does the script generate a string that looks like an identifier? Is this ID stored in a cookie or in any cache mechanisms of the browser?
- **Sending information to a remote address** Are there any pieces of data containing device-specific information sent to a remote server?
- **Minification and Obfuscation** "Minifying" a script consists in removing all unnecessary characters from its source code like white space characters, new line characters or comments without changing its functionality. A lot of well-known JavaScript libraries are "minified" to reduce the amount of data that needs to be transferred when they are downloaded. For example, the weight of the famous jQuery library [105] in version 3.2.0 is cut in three just by minifying the code (from 267.7kb to 86.6kb). Figure 6.1 shows a simple implementation of the Fibonacci sequence in JavaScript. The minified version on the right is much more compact than the one on the left.

  On top of minification, a JavaScript file can be obfuscated, i.e. modified to make it difficult to read and understand. Some variables can be renamed to very short and meaningless names. Some sections can be intertwined to make it difficult to follow the flow of the program. Some parts of the code can also self-generate the true payload similar to what is observed with packing mechanisms in malwares. Most developers use obfuscation to protect their source code and to prevent other developers from copying it but others see it as a way to hide the true meaning of their code. In the end, it requires reverse-engineering efforts to know the true intent of the author and it requires far more means to correctly find if a script is conducting fingerprinting activities.

```
1 function fib(n) {
2     if(n <= 1) {
3         return n;
4     } else {
5         return fib(n - 1) + fib(n - 2);
6     }
7 }
```

Standard

```
1 function fib(a){return a<=1?a:fib
      (a-1)+fib(a-2)}
```

Minified

```
1 eval(function(p,a,c,k,e,d){e=function(c){return c};if(!''.replace(/^/,String)){
      while(c--){d[c]=k[c]||c}k=[function(e){return d[e]}];e=function(){return'\\w+'
      };c=1};while(c--){if(k[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}
      return p}('4 3(0){5 0<=1?0:3(0-1)+3(0-2)}',6,6,'a|||fib|function|return'.split(
      '|'),0,{}))
```

Obfuscated by [108]

Figure 6.1: JavaScript code for the Fibonacci sequence. The three pieces of code are all equivalent.

- **Dividing tasks in different scripts and servers** Instead of collecting attributes in a single script and send everything to a server in a single request, the whole process can be divided in many scripts and the data can be sent to multiple servers behind multiple addresses. This renders the classification more difficult because one script could be considered as benign but, combined with others, it can build a very large fingerprint of any device. To observe these fingerprinting behaviours, an analysis must be performed at the level of a website and not on each script individually. However, this approach presents a new set of challenges as the analysis will have to deal with constant noise from other benign scripts.

In the end, fine tuning all of these rules and identifying a script as a fingerprinting one present many difficulties. Techniques like machine learning could be used to help the classification of scripts

on the web but it remains to be seen how much impact obfuscation and the separation in multiple scripts could have on the results.

6.3.2 Fingerprinting with a very high number of devices

While Panopticlick and AmIUnique brought a much needed insight into the diversity of devices on the web, there remains the question if using fingerprinting for identification at a very large scale is a possibility. Detailed results on this front could nuance the privacy problems raised by this technique as device identification could become impossible when a database reaches a certain number of fingerprints. On a site handling millions of connections every day, is the number of unique fingerprints as high as the ones reported on Panopticlick and AmIUnique? Is the diversity of devices on the web overestimated when extended to millions of users? Unfortunately, these questions can only be answered with a collaboration with a major website as a researchers or organizations like the EFF do not have the reach to bring millions of visitors every day.

6.4 Perspectives and sustainability of browser fingerprinting

Looking at the future of browser fingerprinting yields many questions. The first one is regarding its usage. While researches have proven that a fingerprint can be used to identify a device on the web, it remains to be seen how it can be extended to other purposes. In this thesis, we showed that there is a certain interest from the security community as fingerprinting can be used to fight bots or device impersonation. Yet, there may be hidden benefits of using fingerprinting that are yet to be discovered.

Another important question is the relevance and sustainability of fingerprinting in the continually changing landscape of the web. The actual structure of web browsers gave birth to it: browsers share information about their configuration to improve the user experience. However, is this information still useful today? Could we envision a world where all the device-specific information is not shared publicly? Could we perform all the computations and decisions on the client side? When the web started, the intent behind this information was to help a server deliver tailored web pages to clients. But now that the the web is so rich and dynamic, is it still necessary? If the W3C decides tomorrow with the help of major web actors to reduce or remove the leak of unnecessary device information, they could surely bring the whole browser fingerprinting domain to a halt in less than a year.

Looking at a more distant future, are web browsers destined to disappear? Native applications on smartphones and tablets have seen an incredible rise in popularity in the last decade because they take advantage of the device on which they are installed. However, recent technologies like HTML5 and the introduction of Progressive Web Apps [155] by Google are blurring the line between native apps and mobile websites. Websites now adapt themselves to the widest range of devices possible with techniques like responsive design. We believe that browsers are not going anywhere any time soon as they are still one of the primary conduit to browse content on the web. The diversity that is at the source of fingerprinting is the same one that will make browsers relevant in the 21st century as not everyone will sport an Android or an iPhone device in the future. Then, if browsers are here to stay, in what capacity should browser fingerprinting evolve? We have shown in this thesis that the privacy threats and the security issues coming from fingerprinting are very real. One can learn a lot of information about a device and even on the user behind it. Browser vendors have the power to offer a rich platform for everyone to play with but it must be done with the privacy of all web users in mind. They must act and make extra efforts to give control back in users' hands to prevent issues and abuse in the future. Discussions about browser fingerprinting are also part of a much bigger debate on the current state of online tracking. As a cornerstone of our modern society, the web is a place for everyone and a solution must be found on this tracking front so that the Internet respects all users equally while being sustainable for all actors involved. The future of browser fingerprinting is paved with uncertainty but if it is here to stay, we hope that it will be used in positive ways to make the web safer for everyone.

6.5 Concluding thoughts

To conclude, the future of browser fingerprinting is exciting as we are starting to understand its intricacies. We hope that our work will consolidate the knowledge around this domain and that it will drive the development of new and promising use cases to tackle the security problems of tomorrow. Yet, as this technique depends strongly on the decisions of current web actors and standard organizations, its future is uncertain. Few years from now, will browser fingerprinting exist in the same capacity as it is today? Will browsers share a limited quantity of information to prevent the privacy issues raised in this document? Or is the existence of browser fingerprinting destined to be short-lived as the browser of tomorrow will be fundamentally different from what we know today? At the time of writing, these questions have no answers as they depend on years of discussion, study and development. No matter how the story of browser fingerprinting unfolds in the coming years, there is one certainty: it will never stop to surprise us.

Appendix A

Code snippets from fingerprinting scripts

In this appendix, we show code snippets that are used to collect attributes in fingerprinting scripts. The primary goal of this appendix is to show that many attributes can be collected with very few lines of code without the need to enter complex programming procedure. Comments have been added for some attributes in order to facilitate the understanding of the code.

A.1 Browser APIs

The attributes listed here are collected by calling public interfaces and the language used is JavaScript.

- **List of plugins**: Use of the *navigator* object.

```
//Get the array containing details on all plugins
var np = window.navigator.plugins;
var plist = new Array();
//Go through each Plugin object to collect plugin information
for (var i = 0; i < np.length; i++) {
  //Name of the plugin
  plist[i] = np[i].name + "; ";
  //Description of the plugin
  plist[i] += np[i].description + "; ";
  //Complete name of the file with extension
  //(.dll for Windows, .plugin for Mac, .so for Linux)
  plist[i] += np[i].filename;
  plist[i] += ". ";
}
```

- **Platform**: Use of the *navigator* object.

```
var platform = window.navigator.platform;
```

- **Cookies enabled**: Use of the *navigator* object.

```
var cookieEnabled = window.navigator.cookieEnabled;
```

- **Do Not Track (DNT)**: Use of the *navigator* object.

```
var doNotTrack = window.navigator.doNotTrack;
```

- **Timezone**: Use of a *Date* object.

```
var timezone = new Date().getTimezoneOffset();
```

- **Screen resolution and color depth**: Use of the *screen* object.

```
var resolution = window.screen.width+"x"+window.screen.height+"x"+window.
    screen.colorDepth;
```

- **Local and session storage**: Use of the *localStorage* and *sessionStorage* objects.

```
//We store a value in both localStorage and sessionStorage
try {
    localStorage.fp = "test";
    sessionStorage.fp = "test";
} catch (ex) {
}

//We try retrieving the value we stored previously
//to know if localStorage is enabled
try {
  domLocalStorage = "";
    if (localStorage.fp == "test") {
       domLocalStorage = "yes";
    } else {
       domLocalStorage = "no";
    }
} catch (ex) {
  domLocalStorage = "no";
}

//We try retrieving the value we stored previously
//to know if sessionStorage is enabled
try {
  domSessionStorage = "";
    if (sessionStorage.fp == "test") {
       domSessionStorage = "yes";
    } else {
       domSessionStorage = "no";
    }
} catch (ex) {
  domSessionStorage = "no";
}
```

- **Use of an ad blocker**: We create an element that resembles an ad to see if it is blocked or not.

```
//We create a new division in a script called 'advert.js'
var ads = document.createElement("div");
ads.setAttribute("id", "ads");
document.body.appendChild(ads);

//In fingerprinting script, we check if the "ads" element was created
//If the element does not exist, it means that the script was blocked by an
    extension
//If it is present, the script was executed correctly
var adblocker = document.getElementById('ads')? 'no' : 'yes';
```

- **WebGL Vendor and renderer**: Use of the WebGL API to collect information on the underlying hardware of the device.

```
var canvas = document.createElement('canvas');
//We get a WebGL drawing context to get access to the functions of the API
var ctx = canvas.getContext("webgl") || canvas.getContext("experimental-webgl"
    );

//If the debug extension is present, we collect the information
//If not, we say it is not supported
if(ctx.getSupportedExtensions().indexOf("WEBGL_debug_renderer_info") >= 0) {
  var debugExt = ctx.getExtension('WEBGL_debug_renderer_info');
  webGLVendor = ctx.getParameter(debugExt.UNMASKED_VENDOR_WEBGL);
  webGLRenderer = ctx.getParameter(debugExt.UNMASKED_RENDERER_WEBGL);
} else {
```

```
  webGLVendor = "Not supported";
  webGLRenderer = "Not supported";
}
```

- **Battery**: Use of the now-deprecated Battery API.

```
function updateBatteryUI(battery) {
  //Get the level of charge of the battery between 0 and 1
  var level = (battery.level * 100) + '%';

  //Get the time remaining for the battery to be fully charged
  //If battery is discharging, indicate Infinity
  var cTime = battery.chargingTime + ' Seconds';

  //Get the time remaining for the battery to be empty
  //If battery is charging, indicate Infinity
  var dTime = battery.dischargingTime + ' Seconds';

  if (battery.charging === true) {
    chargingStatus = 'Charging';
  } else if (battery.charging === false) {
    chargingStatus = 'Discharging';
  }
}
```

A.2 Flash

The attributes listed here are collected through Flash. The code listed for each attribute is written in ActionScript [15] and the Flash functions are called directly from JavaScript with the help of the *ExternalInterface* class which "enables straightforward communication between ActionScript and the SWF container" [92].

- **List of fonts**: Use of the *Capabilities* class.

```
var fontNames:Array = [];
for each (var font:Font in Font.enumerateFonts(true)){
  fontNames.push(font.fontName);
}
```

- **Screen resolution**: Use of the *Capabilities* class.

```
var width = Capabilities.screenResolutionX;
var height = Capabilities.screenResolutionY;
```

- **Language**: Use of the *Capabilities* class.

```
var language = Capabilities.language;
```

- **Platform**: Use of the *Capabilities* class.

```
var os = Capabilities.os;
```

Appendix B

Appendices to the AmIUnique study

B.1 Additional data on all AmIUnique's attributes

Table B.1: Normalized Shannon's entropy for all AmIUnique's attributes

Attribute	All	Desktop	Mobile
User agent	0.580	0.550	0.741
List of plugins	0.656	0.718	0.081
List of fonts (Flash)	0.497	0.548	0.033
Screen resolution (JS)	0.290	0.263	0.366
Timezone	0.198	0.200	0.245
Cookies enabled	0.015	0.016	0.011
Accept	0.082	0.082	0.105
Content encoding	0.091	0.089	0.122
Content language	0.351	0.344	0.424
List of HTTP headers	0.249	0.247	0.312
Platform (JS)	0.137	0.110	0.162
Do Not Track	0.056	0.057	0.058
Use of local storage	0.024	0.023	0.036
Use of session storage	0.024	0.023	0.036
Canvas	0.491	0.475	0.512
Vendor WebGL	0.127	0.125	0.131
Renderer WebGL	0.202	0.205	0.165
AdBlock	0.059	0.060	0.029

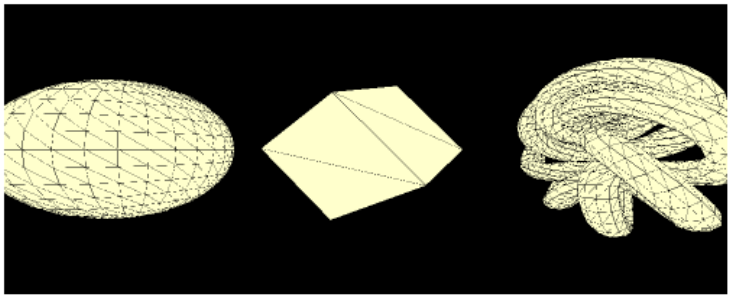
B.2 Our attempt at a WebGL test

As reported by Mowery et al. [116], the WebGL API can be used to render 3D forms in the browser. With the help of the three.js JavaScript library [179], we aimed to have a test that renders three different forms:

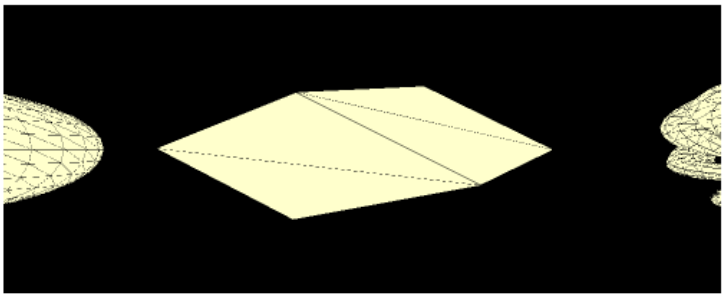
- a sphere
- a cube
- a Torus knot

However, after analyzing more than 40,000 fingerprints, we concluded that the test was too brittle and unreliable to draw any conclusions from it. Indeed, if the user were to change the size of its browser window or open the browser console, the actual dimensions of the rendering context

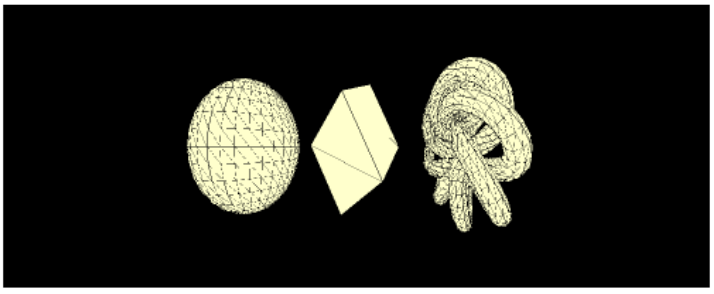
would be updated inside the library and the rendering would differ with just a simple page reload. Figure B.1 shows three renderings of the same test with three different window sizes on the same device.



(a) 1920x1200 window



(b) 960x1200 window



(c) 1080x600 window

Figure B.1: Different renderings of the WebGL test on the same device

B.3 Additional Flash attributes

For Flash, we also collected the following four attributes:

- Capabilities.language
- Capabilities.os
- Capabilties.screenResolutionX
- Capabilties.screenResolutionY

The language obtained through Flash is the devices main language, but it is not as precise as the content language header collected through HTTP. For the screen resolution, it can be more interesting than the JavaScript value because Flash will return the full resolution of a multi-screen setup and not the resolution of a single screen. Finally, when analyzing the data from the string

collected from the OS property, it confirmed what has been observed by Nikiforakis et al. [124] in 2013. Depending on the OS and the browser, the information is often generic, returning "Windows" or "Linux", but in some cases it returns the type of the OS with the exact version of the kernel (for example, "Mac OS 10.8.2" or "Linux 3.18.4-1-ARCH"). This level of detail could be used to forge an attack against a vulnerable system, and it is surprising that little has changed since it was originally reported. In the end, we did not keep this information for our study because it did not increase the number of unique fingerprints and would mainly serve to detect inconsistencies (e.g., caused by User-Agent spoofers).

Table B.2: Statistics of additional Flash attributes

Flash attribute	Distinct values	Unique values
Screen resolution XxY	584	329
Language	44	10
Platform	968	483

B.4 Comparison to the Panopticlick study

To complement Section 3.2.3 of our paper that compares our dataset with the one from Panopticlick [72], we recreated the same graphs to show the impact of 5 years of browser development on browser fingerprinting.

B.4.1 Distribution of fingerprints

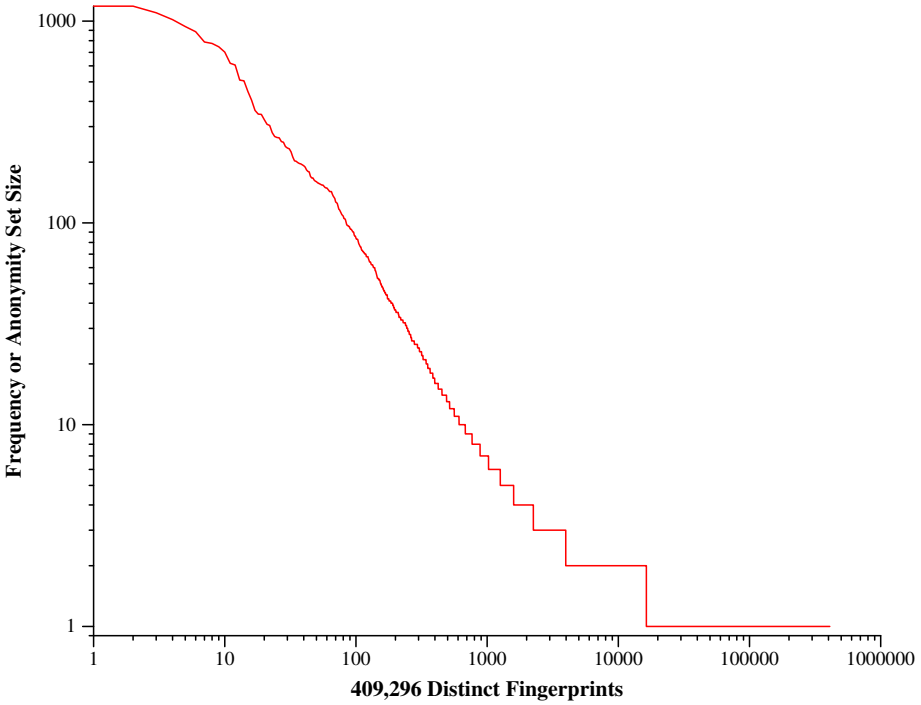
If we compare both frequency distributions in Figure B.2 w.r.t. anonymity set sizes, we can observe that the overall trend is similar in both graphs with set sizes quickly dropping to 1. While Panopticlick has 83.6% of its fingerprints located on the tail on the right of Graph B.2a, AmIUnique presents a slightly lower number on Graph B.2b with 79.4% of fingerprints that are unique in the database (fingerprints with and without JavaScript).

B.4.2 Distribution of browsers

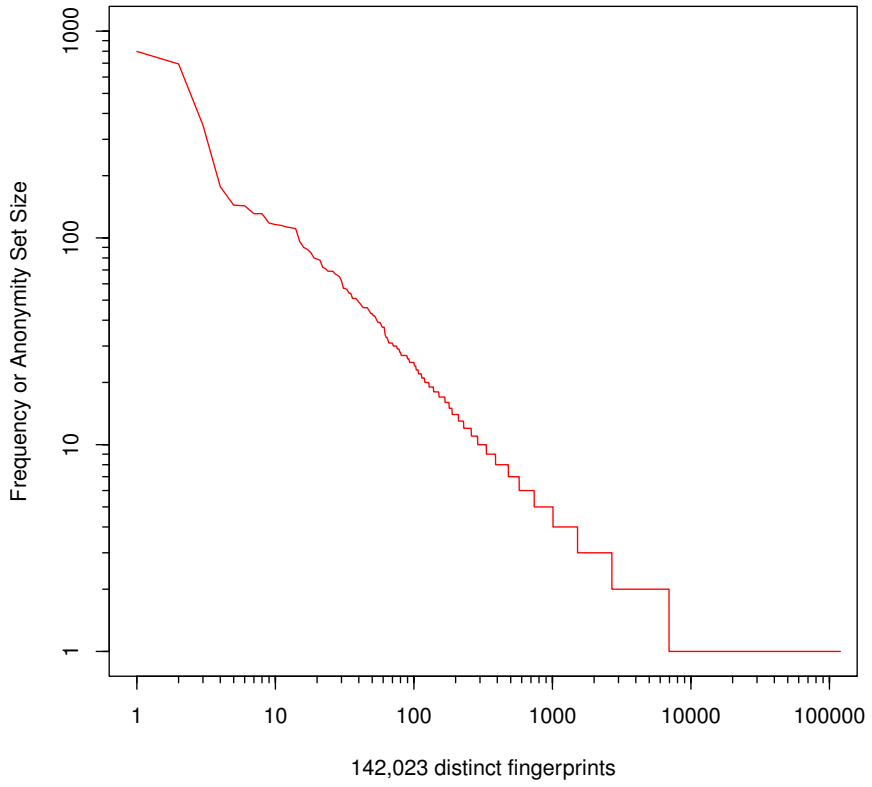
Figure B.3 shows the distribution of surprisal for different categories of browsers. We can see that the overall trend is similar in both graphs. The main noticeable difference is the number of browsers in each category. While the Panopticlick dataset was constituted of mainly Firefox browsers followed by Chrome and Internet Explorer, our dataset put Chrome and Firefox at the same level with all the other browsers behind. This shows the rapid growth of the Chrome userbase over the last 5 years and the decline of Internet Explorer.

B.4.3 Anonymity set sizes

Figure B.4 shows the size of anonymity sets for all attributes if we consider them independently from each other. In our case, the bigger an anonymity set is, the better it is for privacy. If a value is in an anonymity set of size 1, it means that the observed value is unique and is not shared by another fingerprint. With all the attributes that we collected on AmIUnique, we could not add all of them in Figure B.4b for readability reasons so we focused on attributes with the highest level of entropy. If we look at the upper left part of both Figure B.4a and Figure B.4b, we observe very similar results and the most discriminating attributes on AmIUnique are still the same as the ones observed by Eckersley (mainly fonts and plugins) but with the addition of new efficient techniques like canvas fingerprinting.

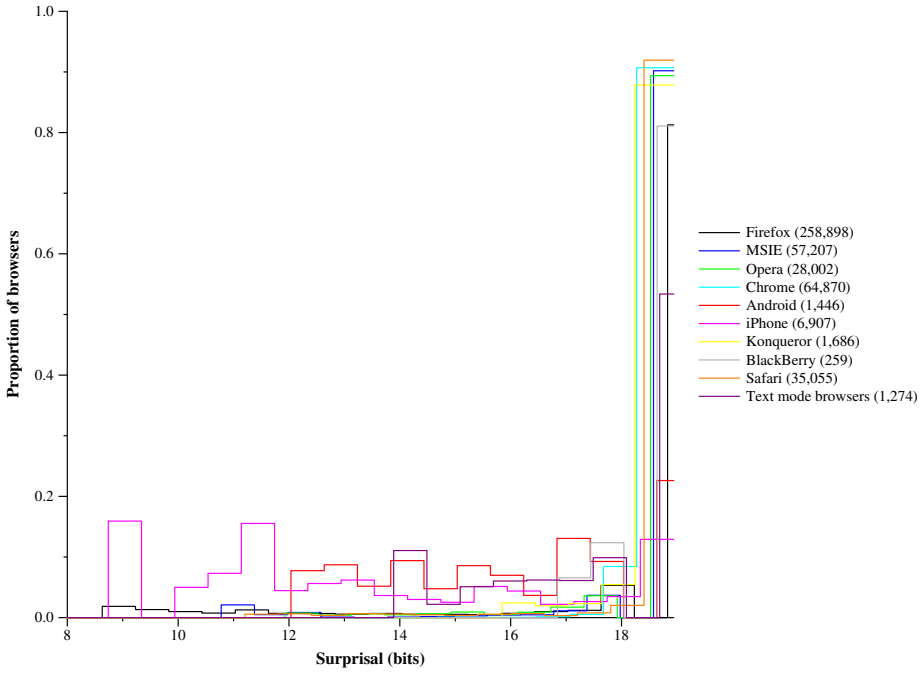


(a) Panopticlick distribution (Fig. 1 of [72])

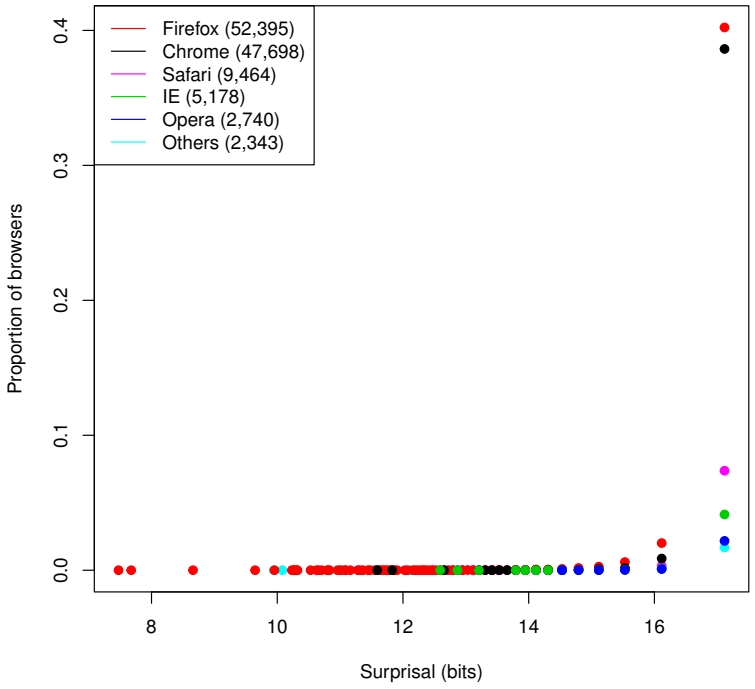


(b) AmIUnique distribution

Figure B.2: Distribution of fingerprints w.r.t. anonymity set size

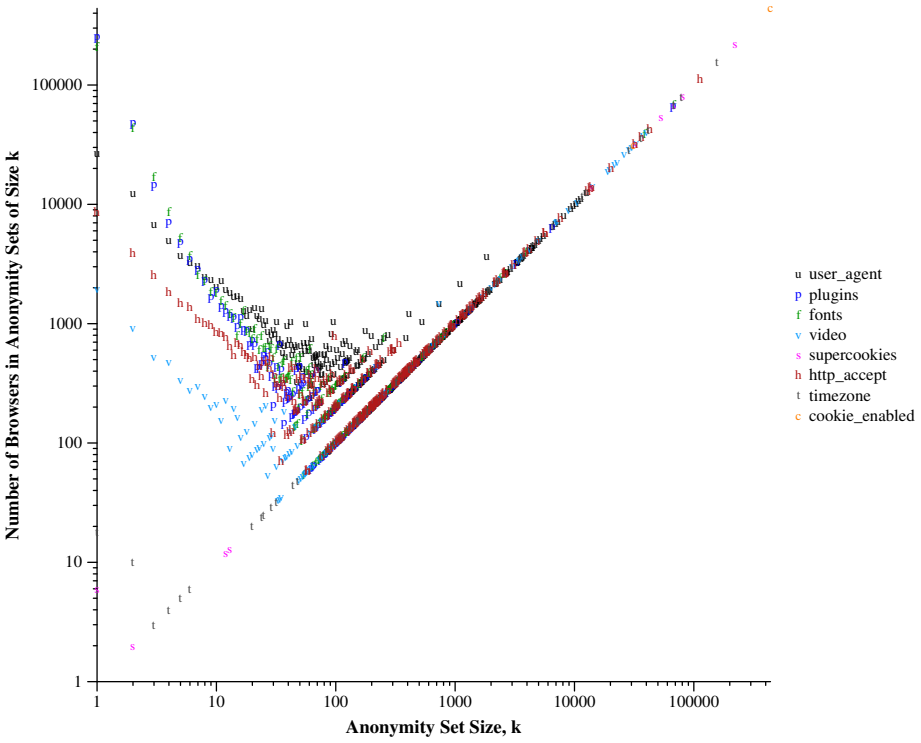


(a) Panopticlick distribution (Fig. 2 of [72])

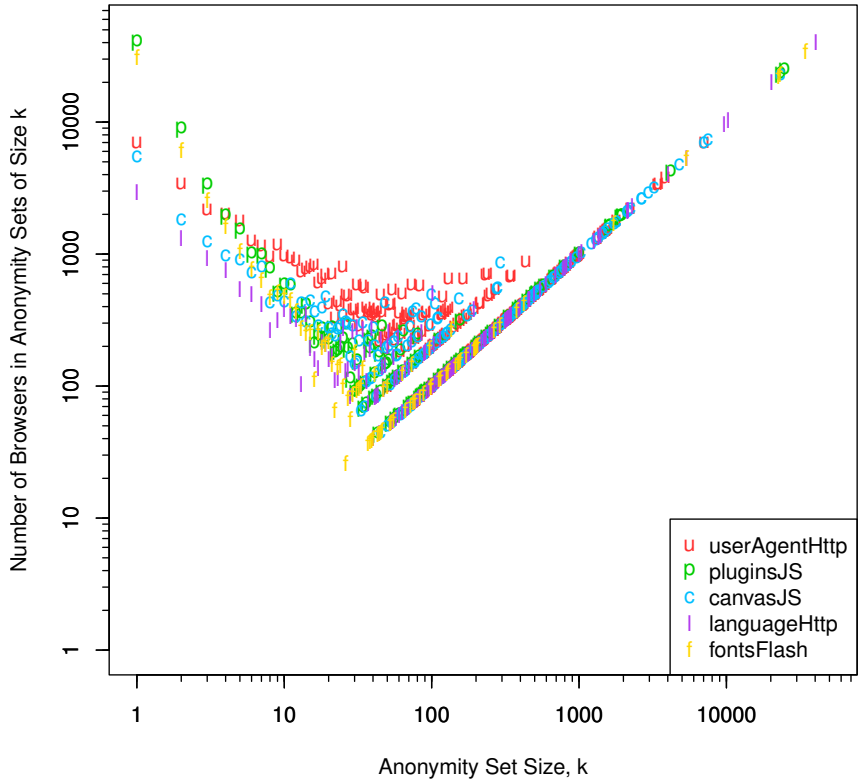


(b) AmIUnique distribution

Figure B.3: Surprisal distributions for different categories of browser



(a) Panopticlick distribution (Fig. 3 of [72])



(b) AmIUnique distribution

Figure B.4: Number of users in anonymity sets of different sizes, considering each variable separately

B.5 Update to the 2016 study

As of June 7th, 2017, we collected 390,410 fingerprints and 316,372 of them had JavaScript activated. We provide here an update of our 2016 study to see which results still hold and which ones have evolved. Table B.3 reports on descriptive statistics of the AmIUnique dataset and Table B.3 gives an update on the number of unique fingerprints in our database

Table B.3: Descriptive statistics of the AmIUnique dataset in February 2016 and June 2017

	February 2016			June 2017			Trend
Attribute	Distinct	Unique	Entropy	Distinct	Unique	Entropy	Entropy
User agent	11,237	6,559	0.580	28,725	17,668	0.601	↗
Accept	131	62	0.082	242	124	0.074	↘
Content encoding	42	11	0.091	91	38	0.123	↗
Content language	4,694	2,887	0.351	9,289	5,569	0.328	↘
List of plugins	47,057	39,797	0.656	72,762	59,443	0.523	↘↘
Cookies enabled	2	0	0.015	3	1	0.014	↘
Use of local/session storage	2	0	0.024	2	0	0.023	↘
Timezone	55	6	0.198	98	27	0.187	↘
Screen resolution	2,689	1,666	0.290	6,573	4,147	0.276	↘
List of fonts	36,202	31,007	0.497	66,822	56,825	0.370	↘↘
List of HTTP headers	1,182	525	0.249	2,181	973	0.244	↘
Platform	187	99	0.137	349	212	0.133	↘
Do Not Track	7	0	0.056	7	0	0.053	↘
Canvas	8,375	5,533	0.491	25,943	19,187	0.503	↗
WebGL Vendor	26	2	0.127	43	12	0.137	↗
WebGL Renderer	1,732	649	0.202	3,222	1,134	0.250	↗
Use of an ad blocker	2	0	0.059	2	0	0.054	↘

Table B.4: Percentage of unique fingerprints

Month	All	Desktop	Mobile
February 2016	89.4%	90.0%	81.0%
June 2017	88.2%	89.2%	82.5%

In the end, the new numbers do not present any surprises and confirm our findings from 2016. All attributes present very similar levels of entropy but we can still see a big drop for both the list of plugins and the list of fonts. This is explained by the fact that the NPAPI architecture has been dropped from both Chrome and Firefox as detailed in Section 3.5. For the numbers of unique fingerprints, the percentages are stable for both Desktop and Mobile devices.

Appendix C

Appendices to the Blink study

C.1 Fingerprint attribute weights

Table C.1: Weights for each attribute of a fingerprint

Attribute	Entropy(bits)
User agent	10.0
Accept Header	6.09
Plugins	15.4
Fonts	13.9
Screen resolution	4.83
Timezone	3.04
DOM storage	2.12
Cookies	0.353

C.2 Fingerprint Dissimilarity

This appendix provides the strategy to determine the dissimilarity between the eight fingerprint attributes.

[Attribute 1] User agent

We decompose the user agent attribute into two categories: data related to the browser and its version, and data related to the architecture of the device (32 or 64 bits).

$$F_{br} = \begin{cases} 0 & br.name_{FP_1} = br.name_{FP_2} \ \wedge \\ & br.ver_{FP_1} = br.ver_{FP_2} \\ 0.125 & br.name_{FP_1} = br.name_{FP_2} \ \wedge \\ & br.ver_{FP_1} \neq br.ver_{FP_2} \\ 1 & br.name_{FP_1} \neq br.name_{FP_2} \end{cases}$$

$$F_{archi} = \begin{cases} 0 & archi_{FP_1} = archi_{FP_2} \\ 1 & archi_{FP_1} \neq archi_{FP_2} \end{cases}$$

We grant equal weight to both these categories. The resulting dissimilarity between the user agents in two fingerprints is computed as follows:

$$d(attr_1(FP_1, FP_2)) = 0.5 \times F_{br} + 0.5 \times F_{archi}$$

[Attribute 2] Plugins

Let us consider $LP(FP_1)$, the set of plugins in fingerprint 1 and $LP(FP_2)$, the set of plugins in fingerprint 2. Plugins are stored in a list of tuples where the first element of each tuple is the name of the plugin and the second its version. We define two different sets for plugins that are common to both fingerprints: one where plugins have an identical version ($LP_{=name,=ver}$) and one where plugins have different versions ($LP_{=name,\neq ver}$).
$P \in LP_{=name,=ver}$ if $\exists P_1 \in LP(FP_1) \wedge \exists P_2 \in LP(FP_2)$, $P_1.name = P_1.name \wedge P_1.ver = P_2.ver \wedge P.name = P_1.name = P_2.name$
$P \in LP_{=name,\neq ver}$ if $\exists P_1 \in LP(FP_1) \wedge \exists P_2 \in LP(FP_2)$, $P_1.name = P_1.name \wedge P_1.ver \neq P_2.ver \wedge P.name = P_1.name = P_2.name$
The total number of plugins in both lists is computed as follows (we count the plugins that appear in both lists once):

$$FU = |(LP(FP_1) \setminus (LP(FP_1) \cap LP(FP_2))) \cup LP(FP_2)|$$

We calculate the proportion of plugins that are unique to FP_1 and to FP_2 as:

$$F1 = \frac{|(LP(FP_1) \setminus (LP_{=name,=ver} \cup LP_{=name,\neq ver}))|}{FU}$$

$$F2 = \frac{|(LP(FP_2) \setminus (LP_{=name,=ver} \cup LP_{=name,\neq ver}))|}{FU}$$

We get the proportion of common plugins:

$$F3 = \frac{|LP_{=name,\neq ver}|}{FU} \qquad F4 = \frac{|LP_{=name,=ver}|}{FU}$$

The dissimilarity between the plugins in two fingerprints is computed as follows:

$$d(attr_2(FP_1, FP_2)) = \frac{F1 + F2 - 0.75 \times F3 - F4 + 1}{2}$$

The dissimilarity for this attribute ranges from 0 to 1 with greater values representing an increasing dissimilarity between the two lists of plugins.

[Attribute 3] Fonts

The dissimilarity between two lists of fonts is the proportion of fonts that are only in one fingerprint, minus the proportion of fonts that are in both. This term is 0 if both lists are identical and 1 if they are completely different.

Let us consider $LF(FP_1)$, the set of fonts in fingerprint 1 and $LF(FP_2)$, the set of fonts in fingerprint 2. The total number of fonts in both lists is computed as follows (we count the fonts that appear in both lists only once):

$$FU = |(LF(FP_1) \setminus (LF(FP_1) \cap LF(FP_2))) \cup LF(FP_2)|$$

The proportion of fonts unique to FP_1 is:

$$F1 = \frac{|(LF(FP_1) \setminus (LF(FP_1) \cap LF(FP_2)))|}{FU}$$

The proportion of fonts unique to FP_2 is:

$$F2 = \frac{|(LF(FP_2) \setminus (LF(FP_1) \cap LF(FP_2)))|}{FU}$$

The proportion of common fonts is:

$$F3 = \frac{|(LF(FP_1) \cap LF(FP_2))|}{FU}$$

The dissimilarity between the fonts in two fingerprints is:

$$d(attr_3(FP_1, FP_2)) = \frac{F1 + F2 - F3 + 1}{2}$$

[Attributes 4 - 8] Accept Header, Screen resolution, Timezone, DOM storage, Cookies

These attributes can be compared without additional processing. The dissimilarity between these attributes is:

$$d(attr_X(FP_1), attr_X(FP_2)) = \begin{cases} 0 & attr_X(FP_1) = attr_X(FP_2) \\ 1 & attr_X(FP_1) \neq attr_X(FP_2) \end{cases}$$

Appendix D

Appendices to the FPRandom study

D.1 Analyzing differences in the AudioContext API

In order to have a better understanding of the diversity of audio fingerprints on the web, we deployed the AudioContext script found by Englehardt et al. on the AmIUnique.org website. After discarding more than 1,000 fingerprints from browsers that did not implement the AudioContext API, we collected in total 19,468 audio fingerprints on a period of 100 days between June and September 2016. The results of this study can be found in Table D.1. We use the Shannon entropy in bits to better represent the probability distribution of each of the attributes. The higher the entropy is, the more diversity is exhibited between devices.

Table D.1: Study of 19,468 audio fingerprints

Name	Entropy (bits)	Size of the biggest set	Number of distinct values	Number of unique values
acSampleRate	1.18	9549	10	3
acState	0.99	10821	2	0
acMaxChannelCount	0.38	18580	11	1
acNumberOfInputs	0.0	19468	1	0
acNumberOfOutputs	0.0	19468	1	0
acChannelCount	0.0	19468	1	0
acChannelCountMode	0.0	19468	1	0
acChannelInterpretation	0.0	19468	1	0
anFftSize	0.0	19468	1	0
anFrequencyBinCount	0.0	19468	1	0
anMinDecibels	0.0	19468	1	0
anMaxDecibels	0.0	19468	1	0
anSmoothingTimeConstant	0.0	19468	1	0
anNumberOfInputs	0.0	19468	1	0
anNumberOfOutputs	0.0	19468	1	0
anChannelCount	0.99	10821	2	0
anChannelCountMode	0.0	19468	1	0
anChannelInterpretation	0.0	19468	1	0
audioDynSum	3.28	5698	53	5
audioDynHash	3.43	5697	72	12

Most of the collected attributes have a single value and do not provide any ground to distinguish one device from another. From the collected audio fingerprints, only 3 attributes have an entropy superior to a single bit:

- *acSampleRate* is the default sample rate of a created track when using the *AudioContext* API. The most common values are 44,1kHz (49,0% of collected fingerprints) and 48kHz (48,5%) but some browsers still present some unusual ones (1,7% have 192kHz and 0,7% 96kHz).
- *audioDynSum* is the sum of 500 frames generated by a very specific audio processing (compressed audio from an oscillator). The precision of each frame is up to 15 decimal digits. The large majority of values are really close to each other with differences only appearing from the 6th or 7th decimal digit.
- *audioDynHash* is similar to *audioDynSum* as it takes the exact same output but it covers the entirety of the rendered track instead of a few hundred frames. As it covers a larger space, the entropy is a little higher and this test exhibits more diversity than all other collected attributes.

With these results, we decided to focus only on the differences created by the audio processing performed inside audio nodes. Especially, we want to introduce random noise in the computed frames so that each run of the same test produces different variations. Other values like the default sample rate are still interesting to change but they can easily be modified and they are not the focus of this work.

D.2 Example of string comparison when ordering JavaScript properties

Figure D.1 illustrates the comparison mechanism between the *appVersion* and the *appName* strings. The engine starts with the 'a' letter on both strings. Translating this letter to their corresponding Latin-1 code points yields the decimal numbers '97'. Subtracting 97 from 97 results in 0. As no difference is detected, the engine continues but faces the exact same result for both the second and third characters in each string as they are identical 'p' letters. However, the behaviour is different from the fourth character. The first string presents a 'V' and the second an 'N'. Translating to their decimal code points yields '86' and '78'. This time, since the subtraction $86 - 78 = 8$ does not give a zero, it informs the engine that a difference has been detected. As the result is positive, *appName* is placed before *appVersion*. If the result of the subtraction were to be negative, it would have been the opposite order.

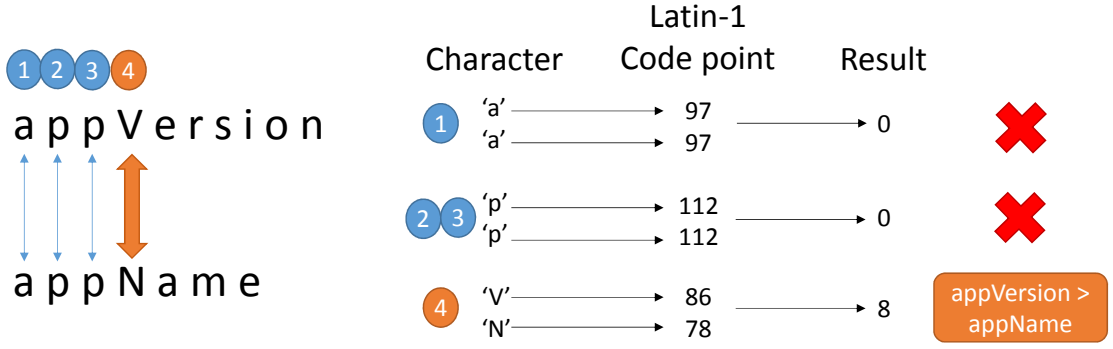


Figure D.1: String comparison between the *appName* and *appVersion* properties

Appendix E

Appendices to the authentication chapter

E.1 Phase 1 of the canvas API study

The first phase of the canvas API study consisted of 15 tests designed to find distinctive features in a canvas rendering. We tried different shapes with different effects for geometric figures and we modified several parameters like the size, the color or the use of stroke for the rendering of strings. The entropy for each test is given in bits. If every collected fingerprint presents a unique rendering, the maximum observable entropy for a test from our dataset of more than 15,000 fingerprints is 13.87 bits. It should be noted that image sizes are not representative of their actual sizes as they have been reduced to fit this page.

Test	Entropy in bits	Real size (WxH)
1	6.87	278x27
2	6.99	389x43
3	7.19	361x41
4	6.87	358x50
5	6.09	153x45
6	6.52	217x60
7	7.30	400x327
8	4.15	400x60
9	2.32	201x60
10	4.89	80x80
11	4.14	146x149
12	4.66	177x179
13	6.85	171x157
14	5.46	259x275
15	4.47	500x500

Tests n°1 to n°7 are renderings of strings with different fonts, colors and sizes. The string in Test n°6 is rendered with a stroke (only the contours are defined). Looking at the entropy, it is slightly lower than Test n°5 so the use of stroke should be avoided. Test n°7 is the same as the 6th one but bigger. The entropy gives a clear signal that size does matter and a clear and well-defined figure definitely helps in distinguishing devices.

Tests n°8 to n°15 are different attempts to draw a wide range of figures. The entropy of all these tests are lower than the ones with a string except for two: Test n°14 with a gradient and Test n°13 which combines both a gradient and a string with an exclamation mark. Depending on the device, the transition between colors can be smooth and natural or it can be clearly marked with a grainy texture on the whole surface.

In the end, a large image with both a string and a gradient appears in our analysis as the right combination in a test to distinguish the maximum number of devices.

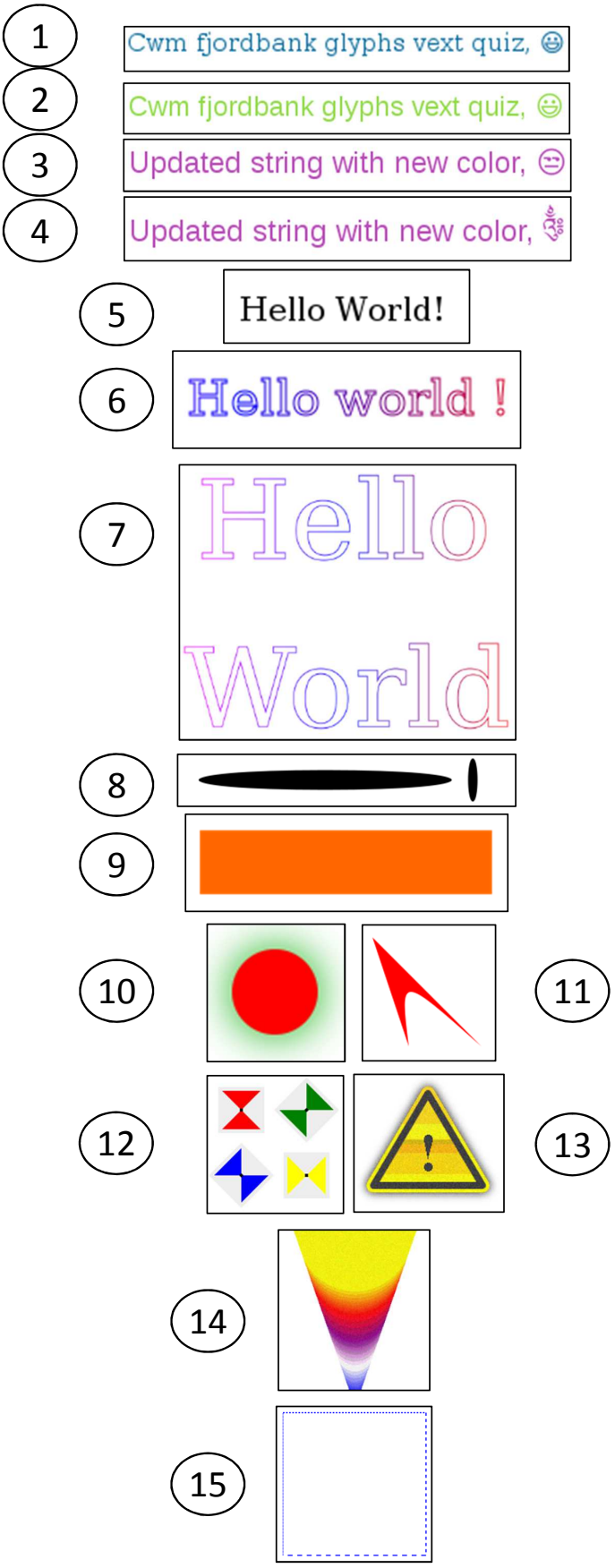


Figure E.1: Phase 1 of canvas tests

E.2 Phase 2 of the canvas API study

The second phase of our study also consisted of 15 tests but the focus was put on the types of renderings observed in the first phase that provided the most diversity between devices. We studied variations on rendered strings and on gradients. If every collected fingerprint presents a unique rendering, the maximum observable entropy for a test from the dataset of this second phase of more than 21,000 fingerprints is 14.37 bits.

Test	Entropy in bits	Real size (WxH)
1	7.69	400x90
2	7.15	400x90
3	7.76	400x90
4	7.73	535x90
5	8.32	800x180
6	7.66	800x180
7	7.21	400x90
8	7.64	400x90
9	7.68	400x90
10	7.70	400x90
11	7.69	400x90
12	7.70	400x90
13	7.74	400x90
14	7.59	400x90
15	7.69	400x90

Test n°1 is the base rendering for our second phase: a string with an emoji and a radial gradient. Test n°2 applies a stroke effect to the rendered string. Test n°3 renders the exact same string as the base one but in a reversed order. The goal of Test n°4 is to study the impact of strings' length. Here, a longer string does not necessarily mean a higher entropy. Test n°5 doubles the size of the string of the base rendering. Bigger strings lead to more differences between devices making the entropy of this test the highest of this second phase. Test n°6 combines a bigger string with a stroke effect. Test n°7 applies a rotation to the rendered string. Test n°8 changes the fallback font to the Arial font which is common in modern operating systems. Test n°9 changes the three colors used in the gradient of the base rendering. Test n°10 changes the position of the three colors of the base gradient. Test n°11 increases the number of colors in the gradient. Test n°12 switches from a radial gradient to a linear gradient. Test n°13 tries a more complex linear gradient with a higher number of points. Test n°14 changes the position of the radial gradient in the canvas element. Test n°15 switches the emoji of the base test to another standard one.

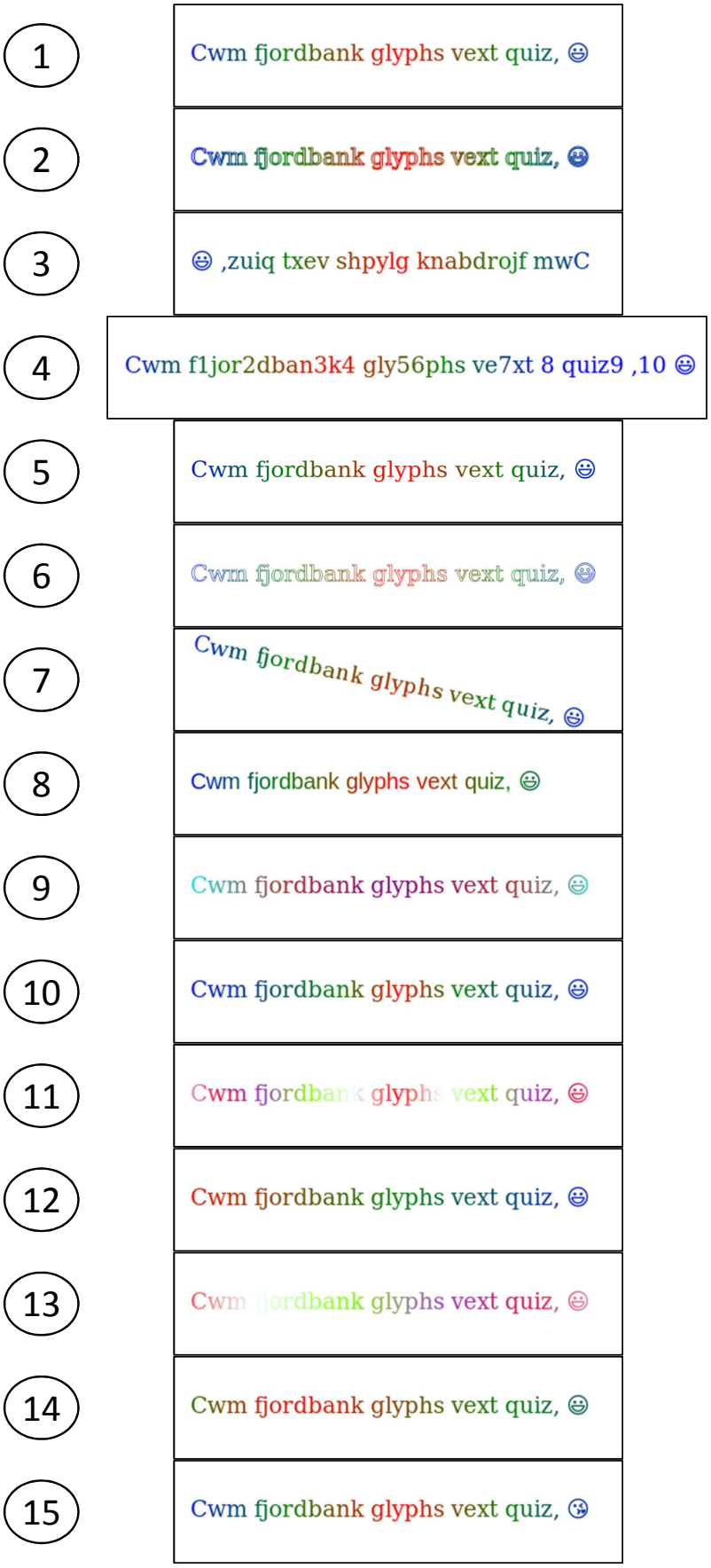


Figure E.2: Phase 2 of canvas tests

E.3 Phase 3 of the canvas API study

The third and final phase of our study consisted of 10 tests to further enhance the diversity between devices. We focused on analysing the impact of Bezier curves and shadows on the overall entropy. If every collected fingerprint presents a unique rendering, the maximum observable entropy for a test from the dataset of this second phase of more than 23,000 fingerprints is 14.50 bits.

Test	Entropy in bits	Real size (WxH)
1	7.68	400x90
2	7.71	400x90
3	7.70	400x90
4	7.71	400x90
5	7.75	400x90
6	7.81	400x90
7	7.84	400x90
8	7.84	400x90
9	8.05	400x90
10	8.11	400x90

Test n°1 is the base rendering for our final phase: a string with an emoji and a radial gradient. It is identical to the base rendering of the second phase. Test n°2 introduces a quadratic Bezier curve. Test n°3 switches to a cubic Bezier curve. Test n°4 mixes these two curves in a single rendering. Test n°5 investigates the impact of increasing the width of a drawn curve. Test n°6 takes the base rendering and adds a black shadow around the rendered string. Test n°7 is identical to the previous one but with an orange shadow. Test n°8 doubles the strength of the shadow blur (from 5 to 10). Test n°9 is a mix of tests n°2, 3 and 5: a light shadow with two Bezier curves. The final test increases the thickness of the curves.

In the end, shadows and curves have a small impact on the global entropy but they still help to further differentiate devices. This phase also echoes the conclusion of the other two phases: bigger canvas elements increase the entropy of the performed tests. This is logical as large elements have better defined edges that could be lost if reduced to a very small surface of pixels.

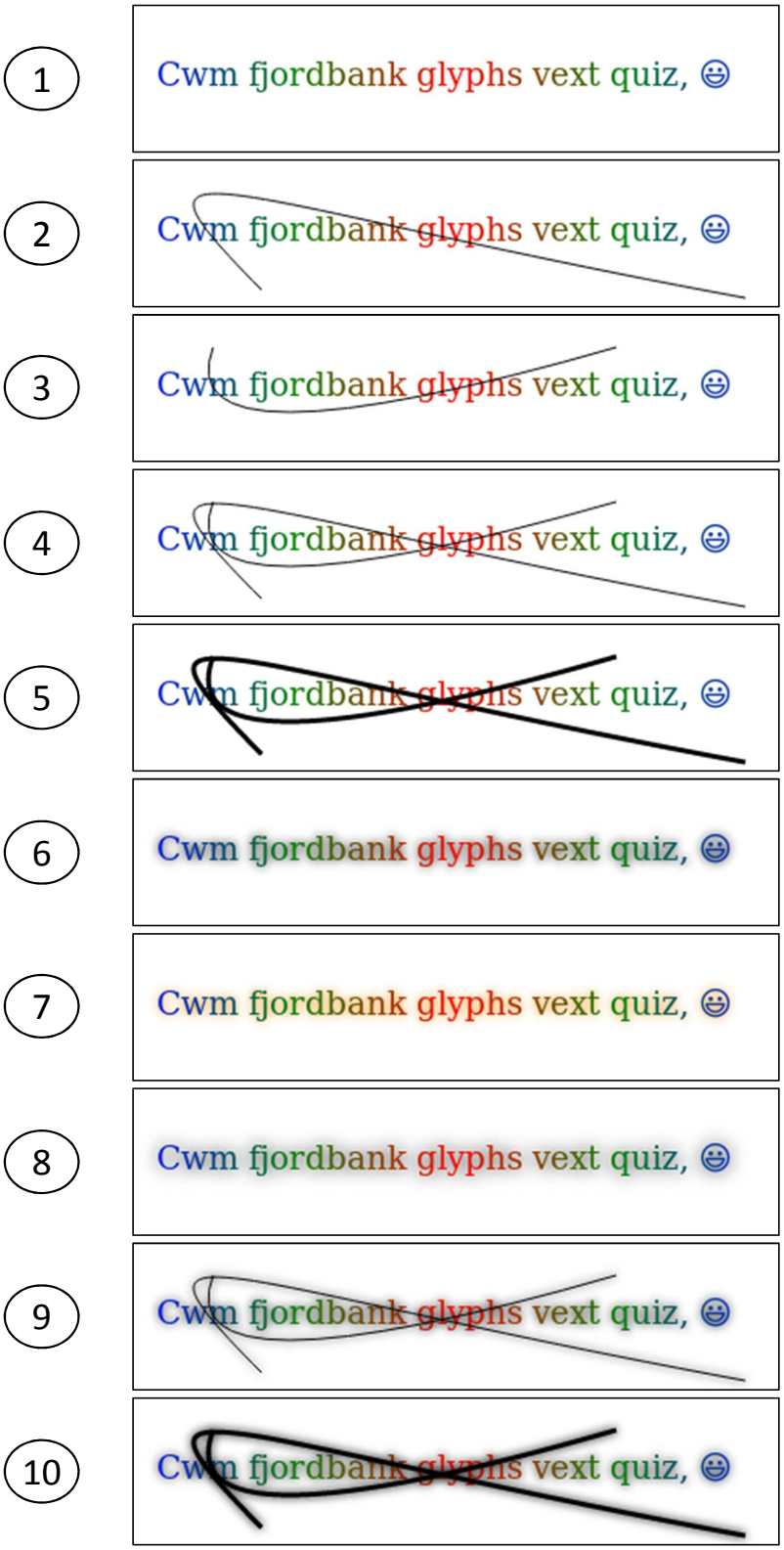


Figure E.3: Phase 3 of canvas tests

E.4 Example of a complete MFA scheme

Figure E.4 gives a concrete example of how our canvas mechanism can be used on top of an already existing authentication scheme to reinforce its security. During the login process, our protocol will be executed as an additional means of protection to make sure that the presented device can produce canvas renderings that match the ones sent during a previous connection. In case of a mismatch, the server will ask for an additional confirmation through a SMS or an app as can be seen in many web services enforcing double-factor authentication. As an added benefit, our mechanism can also be used to prevent connection hijacking by sending new pairs of challenges from time to time to verify the identity of the client.

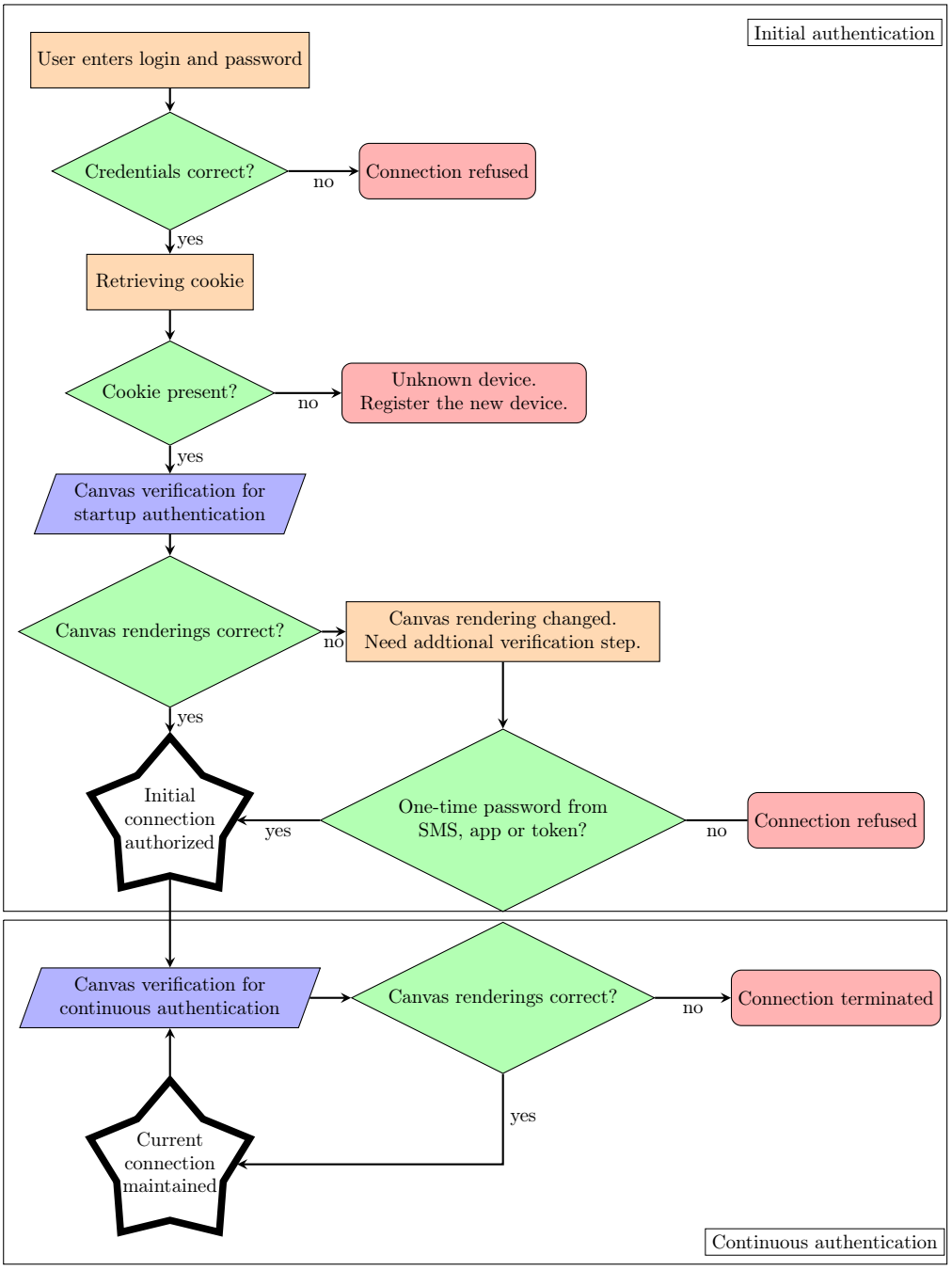


Figure E.4: Example of integration of the canvas mechanism in a complete MFA scheme

List of Figures

2.1 Examples of "Best viewed with" banners . . . 6
2.2 High-level structure of a web browser taken from [44] . . . 7
2.3 Illustration of third-party content delivery . . . 9
2.4 Collection of HTTP headers (passive fingerprinting) . . . 11
2.5 Collection of JavaScript attributes (active fingerprinting) . . . 12
2.6 Excerpt from one script of the *muscial4* malvertising campaign . . . 18
2.7 Examples of canvas renderings on three different devices . . . 23

3.1 Landing page of the AmIUnique.org website . . . 40
3.2 Example of a rendered picture following the canvas fingerprinting test instructions 46
3.3 Comparison of the "Smiling face with open mouth" emoji on different devices and operating systems . . . 47
3.4 Comparison of anonymity set sizes on the list of plugins between desktop and mobile devices . . . 49
3.5 Comparison of anonymity set sizes on the user-agent between desktop and mobile devices . . . 50
3.6 Comparison of anonymity set sizes on the user-agent between Android and iOS devices 51
3.7 Comparison of anonymity set sizes on the user-agent between Chrome and Firefox on mobile devices . . . 51
3.8 Comparison of anonymity set sizes between devices with and without Flash . . . 53
3.9 Evolution of the normalized entropy of plugins for different browsers on desktop computers . . . 54
3.10 Comparison of anonymity set sizes on the complete fingerprint between devices with and without JavaScript . . . 55
3.11 Cumulative distribution function of the elapsed time before a fingerprint evolution 57
3.12 Logo of the Fingerprint Central website . . . 58

4.1 User platform elements involved in web browsing and exhibited in the browser fingerprint . . . 64
4.2 Evolution of the user's platform over time . . . 64
4.3 A multi-level view of browsing platforms. Virtualization isolates the user's system. 68
4.4 An extract of the feature model used for assembling valid DPC configurations . . . 69
4.5 Dissimilarity between consecutive platforms (Leery mode) . . . 71
4.6 Dissimilarity between consecutive platforms (Coffee break mode) . . . 72
4.7 Original canvas rendering with standard colors and the default fallback font . . . 76
4.8 Canvas renderings with modified colors and fonts . . . 76
4.9 Visualization of audio rendered through the AudioContext API . . . 77
4.10 Benchmarking results . . . 79

5.1 Examples of tests with low entropy . . . 85
5.2 Examples of tests with high entropy . . . 86
5.3 Basis of the tests of Phase 2 (Entropy: 7.69 bits) . . . 86
5.4 Zoom on the base rendering with text stroke (Entropy: 7.15 bits) . . . 87
5.5 New effects from Phase 3 (Entropy: 8.11 bits) . . . 87

5.6 Impact of a canvas poisoner on a rendering 89
5.7 Overview of the authentication protocol 90
5.8 Overview of the integration of the canvas mechanism in a multi-factor authentication scheme 91
5.9 Example of a canvas test 92
5.10 Spacing comparison between fonts 92
5.11 Details of a letter 'T' showing pixelation after rotation 92
5.12 Identical strings with different color gradients 93
5.13 Identical forms with different shadow blurs (strongest blur on the right) 93
5.14 Number of sets containing x equal responses 97
5.15 Distribution of canvas renderings for groups with identical fingerprints 99

6.1 JavaScript code for the Fibonacci sequence. The three pieces of code are all equivalent. 104

B.1 Different renderings of the WebGL test on the same device 112
B.2 Distribution of fingerprints w.r.t. anonymity set size 114
B.3 Surprisal distributions for different categories of browser 115
B.4 Number of users in anonymity sets of different sizes, considering each variable separately 116

D.1 String comparison between the *appName* and *appVersion* properties 124

E.1 Phase 1 of canvas tests 127
E.2 Phase 2 of canvas tests 129
E.3 Phase 3 of canvas tests 131
E.4 Example of integration of the canvas mechanism in a complete MFA scheme 132

List of Tables

2.1 Summary of existing defence solutions. M = Modifying the fingerprint content. M* = Modifying the fingerprint content by switching browsers. U = Universal fingerprint. BS = Blocking Scripts. BA = Blocking APIs. 36

3.1 Browser measurements of AmIUnique fingerprints with an example 42
3.2 Summary of statistics 44
3.3 Normalized entropy for six attributes collected both by Panopticlick and AmIUnique 45
3.4 Number of changes per attribute (fingerprints collected between October 2015 and May 2017) 57

4.1 Changed attributes for example n°1 66
4.2 Changed attributes for example n°2 66
4.3 JetStream benchmark results 80
4.4 Web crawl results 81

5.1 Results on the number of canvas changes for different periods of time 88

B.1 Normalized Shannon's entropy for all AmIUnique's attributes 111
B.2 Statistics of additional Flash attributes 113
B.3 Descriptive statistics of the AmIUnique dataset in February 2016 and June 2017 . 117
B.4 Percentage of unique fingerprints 117

C.1 Weights for each attribute of a fingerprint 119

D.1 Study of 19,468 audio fingerprints 123

Author's publications

[1] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Mitigating browser fingerprint tracking: multi-level reconfiguration and diversification. In *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015)*, Firenze, Italy, May 2015.

[2] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the Beast: Diverting modern web browsers to build unique browser fingerprints. In *37th IEEE Symposium on Security and Privacy (S&P 2016)*, San Jose, United States, May 2016.

[3] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. FPRandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In *9th International Symposium on Engineering Secure Software and Systems (ESSoS 2017)*, Bonn, Germany, July 2017.

Tools and prototypes

[4] AmIUnique: Core website - Learn how identifiable you are on the Internet. https://github.com/DIVERSIFY-project/amiunique.

[5] AmIUnique: Extension for Firefox. https://github.com/plaperdr/amiunique-extension-firefox.

[6] AmIUnique: Extension for Chrome. https://github.com/plaperdr/amiunique-extension-chrome.

[7] Blink on VirtualBox: Tool to mitigate browser fingerprint tracking. https://github.com/DIVERSIFY-project/blink.

[8] Blink on Docker: Tool to mitigate browser fingerprint tracking. https://github.com/plaperdr/blink-docker.

[9] Blink: Firefox extension for profile synchronisation. https://github.com/plaperdr/blink-ups-firefox.

[10] Blink: Chrome extension for profile synchronisation. https://github.com/plaperdr/blink-ups-chrome.

[11] Fingerprint Central: A platform to study browser fingerprinting. https://github.com/plaperdr/fp-central.

[12] FPRandom: A browser to counter advanced fingerprinting techniques. https://github.com/plaperdr/fprandom.

Bibliography

[13] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 674–689, New York, NY, USA, 2014. ACM.

[14] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. FPDetective: dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 1129–1140, New York, NY, USA, 2013. ACM.

[15] ActionScript 3.0 overview. https://www.adobe.com/devnet/actionscript/articles/actionscript3_overview.html.

[16] Adblock Plus Official website. https://adblockplus.org/.

[17] The Facts About Our Use of a Canvas Element in Our Recent R&D Test - AddThis. https://www.addthis.com/blog/2014/07/23/the-facts-about-our-use-of-a-canvas-element-in-our-recent-rd-test/.

[18] Furkan Alaca and P. C. van Oorschot. Device Fingerprinting for Augmenting Web Authentication: Classification and Analysis of Methods. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 289–301, New York, NY, USA, 2016. ACM.

[19] Alexa - The top 500 sites on the web. http://www.alexa.com/topsites.

[20] Indétectable et envahissant : le successeur des cookies est là, le fingerprinting - Clubic. http://www.clubic.com/pro/webmarketing/publicite-en-ligne/actualite-742853-fingerprinting-cookies.html.

[21] Les empreintes de nos navigateurs nous identifient — et si on brouillait les pistes ? - Framablog. https://framablog.org/2014/12/23/si-on-brouillait-les-pistes-avec-amiunique/.

[22] #AmIUnique feed - Twitter. https://twitter.com/search?q=%23AmIUnique&src=typd.

[23] How Identifiable Are You On the Web? - Slashdot. https://yro.slashdot.org/story/14/12/14/1943218/how-identifiable-are-you-on-the-web.

[24] AmIUnique extension - Chrome Web Store. https://chrome.google.com/webstore/detail/amiunique/pigjfndpomdldkmoaiiigpbncemhjeca.

[25] AmIUnique extension - Add-ons for Firefox. https://addons.mozilla.org/firefox/addon/amiunique/.

[26] ANGLE: Almost Native Graphics Layer Engine. https://chromium.googlesource.com/angle/angle.

[27] Web Audio API. https://www.w3.org/TR/webaudio/.

[28] HTML Canvas 2D Context - W3C Recommendation 19 November 2015. https://www.w3.org/TR/2dcontext/.

[29] WebVR draft. https://w3c.github.io/webvr/spec/1.1/.

[30] WebGL - OpenGL ES for the Web. https://www.khronos.org/webgl/.

[31] Benchmarks for the WebAudio API. https://github.com/padenot/webaudio-benchmark.

[32] Mika D Ayenson, Dietrich James Wambach, Ashkan Soltani, Nathan Good, and Chris Jay Hoofnagle. Flash cookies and privacy II: Now with HTML5 and ETag respawning. 2011.

[33] Extensive discussion about reducing the HTML battery API. https://groups.google.com/forum/#!topic/mozilla.dev.webapi/6gLD78z6ASI.

[34] Battery Status Event Specification - W3C Working Draft 26 April 2011. https://www.w3.org/TR/2011/WD-battery-status-20110426/.

[35] Battery Status API - W3C Candidate Recommendation 08 May 2012. https://www.w3.org/TR/2012/CR-battery-status-20120508/.

[36] Battery Status API - W3C Candidate Recommendation 07 July 2016. https://www.w3.org/TR/battery-status/.

[37] Peter Baumann, Stefan Katzenbeisser, Martin Stopczynski, and Erik 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, pages 37–46, New York, NY, USA, 2016. ACM.

[38] HyperText and CERN. https://www.w3.org/Administration/HTandCERN.txt.

[39] Online Privacy Policy - BlueCava. http://bluecava.com/privacy-policy/.

[40] Opt-out preferences - BlueCava. http://bluecava.com/opt-out/.

[41] Károly Boda, Ádám Máté Földes, Gábor György Gulyás, and Sándor Imre. *User Tracking on the Web via Cross-Browser Fingerprinting*, volume 7161 of *Lecture Notes in Computer Science*, pages 31–46. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[42] Brave Official website - Browse faster and safer with Brave. https://brave.com/.

[43] Fingerprinting Protection Mode - Brave browser. https://github.com/brave/browser-laptop/wiki/Fingerprinting-Protection-Mode.

[44] How Browsers Work: Behind the scenes of modern web browsers - HTML5 Rocks. https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/.

[45] Elie Bursztein, Artem Malyshev, Tadek Pietraszek, and Kurt Thomas. Picasso: Lightweight Device Class Fingerprinting for Web Clients. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '16, pages 93–102, New York, NY, USA, 2016. ACM.

[46] CanvasBlocker - Firefox extension to block the Canvas API. https://addons.mozilla.org/fr/firefox/addon/canvasblocker/.

[47] Canvas Defender - Firefox add-on that adds unique and persistent noise to a canvas element. https://addons.mozilla.org/en-US/firefox/addon/no-canvas-fingerprinting/.

[48] Yinzhi Cao, Song Li, and Erik Wijmans. (Cross-)Browser Fingerprinting via OS and Hardware Level Features. In *24nd Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[49] Unique Machine demo from [48]. http://uniquemachine.org/.

[50] Chartbeat | Company. https://chartbeat.com/company/.

[51] Mihai Christodorescu, Matthew Fredrikson, Somesh Jha, and Jonathon Giffin. End-to-End Software Diversification of Internet Services. In *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, pages 117–130, New York, NY, 2011. Springer New York.

[52] ClearKey - iovation. https://www.iovation.com/clearkey.

[53] Customer Authentication Datasheet - iovation. https://www.iovation.com/resources/datasheets/clearkey.

[54] CLIQZ Official website - Secure browser with built-in quick search. https://cliqz.com/.

[55] Cliqz buys Ghostery's consumer operations. https://cliqz.com/en/magazine/press-release-cliqz-acquires-ghostery.

[56] John H. Conway. *On Numbers and Games*. Number 6 in London Mathematical Society Monographs. Academic Press, London-New-San Francisco, 1976.

[57] CookieOk - Accpet all cookies warnings automatically. https://cookiesok.com/.

[58] "I don't care about cookies" Chrome extension - Chrome Web Store. https://chrome.google.com/webstore/detail/i-dont-care-about-cookies/fihnjjcciajhdojfnbdddfaoknhalnja.

[59] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant Systems: A Secretless Framework for Security Through Diversity. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[60] Mozilla CSS Extensions. https://developer.mozilla.org/en-US/docs/Web/CSS/Mozilla_Extensions.

[61] Informative Historical Notes - List of known CSS prefixes by W3C. https://www.w3.org/TR/CSS21/syndata.html#vendor-keyword-history.

[62] Adobe Flash Player: List of security vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-53/product_id-6761/Adobe-Flash-Player.html.

[63] Common Vulnerabilities and Exposures - The Standard for Information Security Vulnerability Names. https://cve.mitre.org/.

[64] Desktop Operating System Market Share (April 2017) - NetApplications. https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0&qptimeframe=M&qpsp=219.

[65] Desktop Windows Versions Market Share Worldwide (April 2017) - StatCounter. http://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide.

[66] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[67] Disconnect Official website. https://disconnect.me/.

[68] What is Docker? - Docker Official website. https://www.docker.com/what-docker.

[69] Self-Remediation: Trusted Access from Duo Security - Duo Security. https://duo.com/product/trusted-devices/self-remediation.

[70] Detecting Out of Date and Vulnerable Flash Versions on Your Network - Duo Security. https://duo.com/blog/detecting-out-of-date-and-vulnerable-flash-versions-on-your-network.

[71] EasyList filter lists. https://easylist.to/.

[72] Peter Eckersley. How Unique is Your Web Browser? In *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, PETS'10, pages 1–18, Berlin, Heidelberg, 2010. Springer-Verlag.

[73] ECMA-262, 1st edition, June 1997. https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf.

[74] ECMA-262, 7th edition, June 2016. http://www.ecma-international.org/ecma-262/7.0/index.html.

[75] Emoji and Dingbats. http://unicode.org/faq/emoji_dingbats.html.

[76] Bugzilla - Bug 1231701: Ship an emoji font on Windows XP-7 . https://bugzilla.mozilla.org/show_bug.cgi?id=1231701.

[77] Steven Englehardt and Arvind 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, pages 1388–1401, New York, NY, USA, 2016. ACM.

[78] Cookies - Legal requirements from the European Commission. http://ec.europa.eu/ipg/basics/legal/cookies/index_en.htm.

[79] Opinion 9/2014 on the application of Directive 2002/58/EC to device fingerprinting - ARTICLE 29 DATA PROTECTION WORKING PARTY. http://ec.europa.eu/justice/data-protection/article-29/documentation/opinion-recommendation/files/2014/wp224_en.pdf.

[80] Proposal for a Regulation of the European Parliament and of the Council concerning the respect for private life and the protection of personal data in electronic communications and repealing Directive 2002/58/EC (Regulation on Privacy and Electronic Communications). http://ec.europa.eu/newsroom/dae/document.cfm?doc_id=41241.

[81] Facebook Login documentation. https://developers.facebook.com/docs/facebook-login.

[82] Share Button - Facebook for developers. https://developers.facebook.com/docs/plugins/share-button.

[83] Amin FaizKhademi, Mohammad Zulkernine, and Komminist Weldemariam. FPGuard: Detection and Prevention of Browser Fingerprinting. In *Data and Applications Security and Privacy XXIX*, volume 9149 of *Lecture Notes in Computer Science*, pages 293–308. Springer International Publishing, 2015.

[84] A New Way to Control the Ads You See on Facebook, and an Update on Ad Blocking - Facebook Newsroom. https://newsroom.fb.com/news/2016/08/a-new-way-to-control-the-ads-you-see-on-facebook-and-an-update-on-ad-blocking/.

[85] FB reblock: ad-blocking community finds workaround to Facebook - Adblock Plus Official website. https://adblockplus.org/blog/fb-reblock-ad-blocking-community-finds-workaround-to-facebook.

[86] David Fifield and Serge Egelman. Fingerprinting web users through font metrics. In *Proceedings of the 19th international conference on Financial Cryptography and Data Security*, Berlin, Heidelberg, 2015. Springer-Verlag.

[87] Anonymous browser fingerprint - fingerprintjs. https://github.com/Valve/fingerprintjs.

[88] Ugo Fiore, Aniello Castiglione, Alfredo De Santis, and Francesco Palmieri. Countering browser fingerprinting techniques: Constructing a fake profile with google chrome. In *Network-Based Information Systems (NBiS), 2014 17th International Conference on*, pages 355–360. IEEE, 2014.

[89] Mozilla Firefox ESR Overview. https://www.mozilla.org/firefox/organizations/faq/.

[90] An Update on Flash Player and Android. https://blogs.adobe.com/flashplayer/2012/06/flash-player-and-android-update.html.

[91] Capabilities - ActionScript® 3 (AS3) API Reference. http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/system/Capabilities.html.

[92] ExternalInterface - ActionScript® 3 (AS3) API Reference. http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/external/ExternalInterface.html.

[93] Fingerprint Central - 2016 - Google Summer of Code Archive. https://summerofcode.withgoogle.com/archive/2016/projects/5915605559410688/.

[94] Device Tracking Add-on for minFraud Services - MaxMind. https://dev.maxmind.com/minfraud/device/.

[95] ThreatMetrix Announces Cookieless Device Identification to Prevent Online Fraud While Protecting Customer Privacy - ThreatMetrix. https://www.threatmetrix.com/press-releases/threatmetrix-announces-cookieless-device-identification-to-prevent-online-fraud-while-prot

[96] Ghostery Official website. https://www.ghostery.com/.

[97] Gábor György Gulyás, Gergely Acs, and Claude Castelluccia. Near-Optimal Fingerprinting with Constraints. In *PET Symposium '16*, Darmstadt, Germany, July 2016.

[98] History of the browser user-agent string. http://webaim.org/blog/user-agent-string-history/.

[99] History of the user-agent string. https://www.nczonline.net/blog/2010/01/12/history-of-the-user-agent-string/.

[100] HTML Canvas 2D Context. http://www.w3.org/TR/2dcontext/.

[101] Yih Huang and Anup K. Ghosh. *Introducing Diversity and Uncertainty to Create Moving Attack Surfaces for Web Services*, pages 131–151. Springer New York, New York, NY, 2011.

[102] The state of the blocked web - 2017 Global Adblock Report by PageFair. https://www.iab.com/wp-content/uploads/2016/04/IAB_Internet_Advertising_Revenue_Report_FY_2016.pdf.

[103] A crash course in just-in-time (JIT) compilers - Mozilla Hacks, the Web developer blog. https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/.

[104] JPEG XR Codec Overview. https://msdn.microsoft.com/en-us/library/windows/desktop/hh707223(v=vs.85).aspx.

[105] jQuery Official Website. https://jquery.com/.

[106] Introducing the JetStream Benchmark Suite. https://webkit.org/blog/3418/introducing-the-jetstream-benchmark-suite/.

[107] JetStream benchmark. http://browserbench.org/JetStream/.

[108] JavaScript Obfuscator. http://www.danstools.com/javascript-obfuscate/index.php.

[109] Sheharbano Khattak, David Fifield, Sadia Afroz, Mobin Javed, Srikanth Sundaresan, Damon McCoy, Vern Paxson, and Steven J. Murdoch. Do You See What I See? Differential Treatment of Anonymous Users. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[110] Andreas Kurtz, Hugo Gascon, Tobias Becker, Konrad Rieck, and Felix Freiling. Fingerprinting mobile devices using personalized configurations. *Proceedings on Privacy Enhancing Technologies*, 2016(1):4–19, 2016.

[111] Masking Agent extension for Firefox. https://addons.mozilla.org/firefox/addon/masking-agent/.

[112] Jonathan R Mayer. Any person... a pamphleteer": Internet Anonymity in the Age of Web 2.0. *Undergraduate Senior Thesis, Princeton University*, 2009.

[113] Georg Merzdovnik, Markus Huber, Damjan Buhov, Nick Nikiforakis, Sebastian Neuner, Martin Schmiedecker, and Edgar Weippl. Block me if you can: A large-scale study of tracker-blocking tools. In *2nd IEEE European Symposium on Security and Privacy, Paris, France*, 2017.

[114] Mobile web browsing overtakes desktop for the first time - The Guardian. https://www.theguardian.com/technology/2016/nov/02/mobile-web-browsing-desktop-smartphones-tablets.

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

[116] Keaton Mowery and Hovav Shacham. Pixel Perfect: Fingerprinting Canvas in HTML5. In Matt Fredrikson, editor, *Proceedings of W2SP 2012*. IEEE Computer Society, May 2012.

[117] Bug 1313580 - Remove web content access to Battery API. https://bugzilla.mozilla.org/showbug.cgi?id=1313580.

[118] Firefox 11 for developers. https://developer.mozilla.org/Firefox/Releases/11.

[119] Firefox 52 Release Notes. https://www.mozilla.org/en-US/firefox/52.0/releasenotes/.

[120] Muhammad Haris Mughees, Zhiyun Qian, Zubair Shafiq, Karishma Dash, and Pan Hui. A First Look at Ad-block Detection: A New Arms Race on the Web. *CoRR*, abs/1605.05841, 2016.

[121] Martin Mulazzani, Philipp Reschl, Markus Huber, Manuel Leithner, Sebastian Schrittwieser, Edgar Weippl, and FH Campus Wien. Fast and reliable browser identification with javascript engine fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP)*, volume 5, 2013.

[122] Gabi Nakibly, Gilad Shelef, and Shiran Yudilevich. Hardware Fingerprinting Using HTML5. *CoRR*, abs/1503.01408, 2015.

[123] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. PriVaricator: Deceiving Fingerprinters with Little White Lies. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 820–830, Republic and Canton of Geneva, Switzerland, 2015. International World Wide Web Conferences Steering Committee.

[124] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 541–555, Washington, DC, USA, 2013. IEEE Computer Society.

[125] Nmap: the Network Mapper - OS Detection. https://nmap.org/book/man-os-detection.html.

[126] NoScript Official website. https://noscript.net/.

[127] NPAPI deprecation: developer guide. https://www.chromium.org/developers/npapi-deprecation — The Netscape Plugin API (NPAPI) has been permanently removed from Google Chrome since version 45. The Pepper API (PPAPI) is one option but few plugins exist and it is not proposed in the developer guide as an alternative.

[128] User Authentication with OAuth 2.0. https://oauth.net/articles/authentication/.

[129] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein. Finding Focus in the Blur of Moving-Target Techniques. *IEEE Security Privacy*, 12(2):16–26, Mar 2014.

[130] Łukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Diaz. *The Leaking Battery*, pages 254–263. Springer International Publishing, Cham, 2016.

[131] Lukasz Olejnik, Steven Englehardt, and Arvind Narayanan. Battery Status Not Included: Assessing Privacy in Web Standards. In *3rd International Workshop on Privacy Engineering (IWPE'17)*, San Jose, United States, 2017.

[132] OpenID Connect official website. https://openid.net/connect/.

[133] Operation Fingerprint - A look into several Angler Exploit Kit malvertising campaigns. https://malwarebytes.app.box.com/v/operation-fingerprint.

[134] Wrong order in Object properties interation - V8 bug tracker. https://bugs.chromium.org/p/v8/issues/detail?id=164.

[135] PageFair Official website. https://pagefair.com/.

[136] The state of the blocked web - 2017 Global Adblock Report by PageFair. https://pagefair.com/downloads/2017/01/PageFair-2017-Adblock-Report.pdf.

[137] The cost of ad blocking - PageFair and Adobe 2015 Ad Blocking Report. https://downloads.pagefair.com/wp-content/uploads/2016/05/2015_report-the_cost_of_ad_blocking.pdf.

[138] Pale Moon browser - Version 25.6.0 adds a canvas poisoning feature. https://www.palemoon.org/releasenotes.shtml.

[139] NPAPI Plugins in Firefox. https://blog.mozilla.org/futurereleases/2015/10/08/npapi-plugins-in-firefox/.

[140] Disallow enumeration of navigator.plugins (Mozilla bug tracker). https://bugzilla.mozilla.org/show_bug.cgi?id=757726.

[141] Cross-browser plugin detection - Windows Developer Network. https://msdn.microsoft.com/library/dn423948(v=vs.85).aspx.

[142] Amazon Privacy Notice. https://www.amazon.com/gp/help/customer/display.html?nodeId=468496.

[143] Apple Privacy Policy. http://www.apple.com/legal/privacy/en-ww/.

[144] Privacy Badger Official website - Electronic Frontier Foundation. https://www.eff.org/privacybadger.

[145] Facebook Data Policy [Accessed March 2017]. https://www.facebook.com/full_data_use_policy.

[146] Google Privacy Policy - Archive April 2001. https://www.google.com/policies/privacy/archive/19990920-20010104/.

[147] Google Privacy Policy - Archive July 2004. https://www.google.com/policies/privacy/archive/20010104-20040701/.

[148] Google Privacy Policy - Archive March 2012. https://www.google.com/policies/privacy/archive/20111020-20120301/.

[149] Google Privacy Policy - Archive March 2014. https://www.google.com/policies/privacy/archive/20131220-20140331/.

[150] Google Privacy Policy - Archive June 2015. https://www.google.com/policies/privacy/archive/20150501-20150605/.

[151] Google Privacy Policy. https://www.google.com/policies/privacy/.

[152] Microsoft Privacy Statement. https://privacy.microsoft.com/en-US/privacystatement.

[153] Wikimedia Foundation Privacy Policy [Accessed March 2017]. https://wikimediafoundation.org/wiki/Privacy_policy.

[154] Yahoo Privacy Center. https://privacy.yahoo.com/.

[155] Progressive Web Apps - Google Developers. https://developers.google.com/web/progressive-web-apps/.

[156] A Quantum Leap for the Web - Mozilla Tech. https://medium.com/mozilla-tech/a-quantum-leap-for-the-web-a3b7174b3c12.

[157] Random Agent Spoofer - Firefox extension. https://addons.mozilla.org/firefox/addon/random-agent-spoofer/.

[158] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, May 1998.

[159] fingerprintjs2, modern and flexible browser fingerprinting library, a successor to the original fingerprintjs. https://github.com/Valve/fingerprintjs2.

[160] OpenWPM - A web privacy measurement framework. https://github.com/citp/OpenWPM.

[161] RFC 7932 - Brotli Compressed Data Format. https://tools.ietf.org/html/rfc7932.

[162] RFC 2109 - HTTP State Management Mechanism. https://tools.ietf.org/html/rfc2109.

[163] RFC 2965 - HTTP State Management Mechanism. https://tools.ietf.org/html/rfc2965.

[164] RFC 1866 - Hypertext Markup Language - 2.0. https://tools.ietf.org/html/rfc1866.

[165] RFC 1945 - Hypertext Transfer Protocol – HTTP/1.0. https://tools.ietf.org/html/rfc1945.

[166] RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1. https://tools.ietf.org/html/rfc2616.

[167] T. Saito, K. Takahashi, K. Yasuda, T. Ishikawa, K. Takasu, T. Yamada, N. Takei, and R. Hosoi. OS and Application Identification by Installed Fonts. In *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, pages 684–689, March 2016.

[168] 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)*, pages 587–592, July 2016.

[169] J. Schuh. Saying Goodbye to Our Old Friend NPAPI, September 2013. https://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html.

[170] Device / Browser Fingerprinting - Heuristic-based Authentication - SecurAuth. https://docs.secureauth.com/pages/viewpage.action?pageId=40045162.

[171] Suphannee Sivakorn, Jason Polakis, and Angelos D Keromytis. I'm not a human: Breaking the Google reCAPTCHA. https://www.blackhat.com/docs/asia-16/materials/asia-16-Sivakorn-Im-Not-a-Human-Breaking-the-Google-reCAPTCHA-wp.pdf.

[172] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. Discovering Browser Extensions via Web Accessible Resources. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, pages 329–336, New York, NY, USA, 2017. ACM.

[173] Ashkan Soltani, Shannon Canty, Quentin Mayo, Lauren Thomas, and Chris Jay Hoofnagle. Flash Cookies and Privacy. In *AAAI spring symposium: intelligent information privacy management*, volume 2010, pages 158–163, 2010.

[174] Tracking Preference Expression (DNT) - W3C Candidate Recommendation. https://www.w3.org/TR/tracking-dnt/.

[175] Jan Spooren, Davy Preuveneers, and Wouter Joosen. Mobile Device Fingerprinting Considered Harmful for Risk-based Authentication. In *Proceedings of the Eighth European Workshop on System Security*, EuroSec '15, pages 6:1–6:6, New York, NY, USA, 2015. ACM.

[176] Jan Spooren, Davy Preuveneers, and Wouter Joosen. Leveraging Battery Usage from Mobile Devices for Active Authentication. *Mobile Information Systems*, 2017:1367064:1–1367064:14, 2017.

[177] Oleksii Starov and Nick Nikiforakis. XHOUND: Quantifying the Fingerprintability of Browser Extensions. In *38th IEEE Symposium on Security and Privacy (S&P 2017)*, San Jose, United States, 2017.

[178] GPU accelerating 2D Canvas and enabling 3D content for older GPUs - Chromium blog. https://blog.chromium.org/2012/02/gpu-accelerating-2d-canvas-and-enabling.html.

[179] three.js, a JavaScript library to create 3D animations using WebGL - Official website. https://threejs.org/.

[180] High Resolution Time Level 2 (JavaScript Performance API. https://www.w3.org/TR/hr-time/#dom-domhighrestimestamp.

[181] Tor Browser - Tor Project Official website. https://www.torproject.org/projects/torbrowser.html.

[182] Release of Tor with a new defense against font enumeration. https://blog.torproject.org/blog/tor-browser-55-released.

[183] Tor Browser 6.5 is released - The Tor Blog. https://blog.torproject.org/blog/tor-browser-65-released.

[184] The Design and Implementation of the Tor Browser [DRAFT] - Tor Project Official website. https://www.torproject.org/projects/torbrowser/design/.

[185] Christof Torres, Hugo Jonker, and Sjouke Mauw. FP-Block: usable web privacy by controlling browser fingerprinting. In *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS 2015)*, 2015.

[186] Math routines are OS fingerprintable - Tor bug tracker. https://trac.torproject.org/projects/tor/ticket/13018.

[187] Add an FPCentral test to our test suite - Tor bug tracker. https://trac.torproject.org/projects/tor/ticket/22587.

[188] Desktop and Mobile Ad Revenue Surpasses TV for the First Time. http://adage.com/article/digital/digital-ad-revenue-surpasses-tv-desktop-iab/308808/.

[189] Tweet Button - Twitter Developers. https://dev.twitter.com/web/tweet-button.

[190] Two Factor Auth List - List of websites supporting two-factor authentication and the methods theu use. https://twofactorauth.org/.

[191] This Is Your Brain On Uber. http://www.npr.org/2016/05/17/478266839/this-is-your-brain-on-uber.

[192] uBlock Origin - An efficient blocker for Chromium and Firefox. Fast and lean. https://github.com/gorhill/uBlock.

[193] T. Unger, M. Mulazzani, D. Frühwirt, M. Huber, S. Schrittwieser, and E. Weippl. SHPF: Enhancing HTTP(S) Session Security with Browser Fingerprinting. In *2013 International Conference on Availability, Reliability and Security*, pages 255–261, Sept 2013.

[194] Do Not Track: an uncertain future for the web's most ambitious privacy initiative. http://www.theverge.com/2012/10/12/3485590/do-not-track-explained.

[195] TeamID blog post: "Announcing Our Worst Passwords of 2016". https://www.teamsid.com/worst-passwords-2016/.

[196] History of the Web - World Wide Web Foundation. http://webfoundation.org/about/vision/history-of-the-web/.

[197] WebGL Specification. https://www.khronos.org/registry/webgl/specs/latest/1.0/.

[198] WebGL Extension Registry. https://www.khronos.org/registry/webgl/extensions/.

[199] Bug 164213 - Remove Battery Status API from the tree. https://bugs.webkit.org/show_bug.cgi?id=164213.

[200] Giving Web a Memory Cost Its Users Privacy - The New York Times. http://www.nytimes.com/2001/09/04/business/giving-web-a-memory-cost-its-users-privacy.html.

[201] WebP - A new image format for the Web. https://developers.google.com/speed/webp/.

[202] Codepage layout - ISO/IEC 8859-1. https://en.wikipedia.org/wiki/ISO/IEC_8859-1#Codepage_layout.

[203] W. Wu, J. Wu, Y. Wang, Z. Ling, and M. Yang. Efficient Fingerprinting-Based Android Device Identification With Zero-Permission Identifiers. *IEEE Access*, 4:8073–8083, 2016.

[204] Yahoo Security Notice September 22, 2016. https://help.yahoo.com/kb/sln28092.html.

[205] Yahoo Security Notice December 14, 2016. https://help.yahoo.com/kb/SLN27925.html.

[206] Yahoo reveals more breachiness to users victimized by forged cookies. https://arstechnica.com/information-technology/2017/02/yahoo-reveals-more-breachiness-to-users-victimized-by-forged-cookies/.

[207] Beware Evil APIs. https://browser.yandex.com/blog/beware-evil-apis.

[208] Zhonghao Yu, Sam Macbeth, Konark Modi, and Josep M. Pujol. Tracking the Trackers. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 121–132, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee.

INSA de RENNES
Service des Formations

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

Titre de la thèse:
Browser Fingerprinting: Exploring Device Diversity to Augment Authentification and Build Client-Side Countermeasures

Nom Prénom de l'auteur : LAPERDRIX PIERRE

Membres du jury :
- Monsieur BAUDRY Benoît
- Monsieur CASTELLUCCIA Claude
- Monsieur AVOINE Gildas
- Monsieur PIESSENS Frank
- Monsieur ME Ludovic
- Madame BIELOVA Natalia

Président du jury : Ludovic Mé

Date de la soutenance : 03 Octobre 2017

Reproduction de la these soutenue

☑ Thèse pouvant être reproduite en l'état
☐ Thèse pouvant être reproduite après corrections suggérées

Fait à Rennes, le 03 Octobre 2017

Signature du président de jury

Le Directeur,

M'hamed DRISSI

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES - RENNES

Résumé

L'arrivée de l'Internet a révolutionné notre société à l'aube du 21e siècle. Nos habitudes se sont métamorphosées pour prendre en compte cette nouvelle manière de communiquer et de partager avec le monde. Grâce aux technologies qui en constituent ses fondations, le web est une plateforme universelle. Que vous utilisiez un PC de bureau sous Windows, un PC portable sous MacOS, un serveur sous Linux ou une tablette sous Android, chacun a les moyens de se connecter à ce réseau de réseaux pour partager avec le monde. La technique dite de ***Browser fingerprinting*** est née de cette diversité logicielle et matérielle qui compose nos appareils du quotidien. En exécutant un script dans le navigateur web d'un utilisateur, un serveur peut récupérer une très grande quantité d'informations. Il a été démontré qu'il est possible d'identifier de façon unique un appareil en récoltant suffisamment d'informations. L'impact d'une telle approche sur la vie privée des internautes est alors conséquente, car le *browser fingerprinting* est totalement indépendant des systèmes de traçage connu comme les cookies. Dans cette thèse, nous apportons les contributions suivantes :

1- Grâce au site *AmiUnique.org*, **nous analysons 118,934 empreintes**. Nous démontrons que l'identification d'appareils est toujours possible, car 89.4% des empreintes collectées sur notre site sont uniques. Nous constatons aussi que l'identification d'appareils mobiles est possible, même si ces plateformes logicielles et matérielles sont beaucoup plus restreintes.

2- Nous détaillons **deux contre-mesures appelées Blink et FPRandom**. Modifier artificiellement le contenu d'une empreinte présente de nombreuses difficultés, car un mauvais changement peut empêcher l'utilisateur de naviguer sur Internet. En exploitant la diversité logicielle et en introduisant un comportement aléatoire dans certaines fonctions des navigateurs, nous changeons constamment l'empreinte présentée à un traqueur pour l'empêcher d'identifier un appareil sur Internet.

3- De plus en plus de bases de données sont attaquées chaque année, et une quantité de plus en plus grande de mots de passe se retrouve en libre accès sur Internet. Pour améliorer la sécurité des systèmes d'authentification, nous avons conçu **un protocole d'authentification complet basé sur la *canvas fingerprinting***. En demandant au navigateur de dessiner une image très précise, il nous est possible de vérifier si l'appareil utilisé est connu ou non du système. Notre protocole permet d'éviter des piratages de compte suite à un vol de mot de passe.

Le *browser fingerprinting* est un domaine fascinant qui en est encore à ses balbutiements. Avec cette thèse, nous contribuons à l'écriture des premières pages de son histoire en fournissant une vue d'ensemble du domaine, de ses fondations jusqu'à l'impact des nouvelles technologies du web sur cette technique. Nous nous tournons aussi vers le futur via l'exploration d'une nouvelle facette du domaine afin d'améliorer la sécurité des comptes sur Internet.

Abstract

Users are presented with an ever-increasing number of choices to connect to the Internet. From desktops, laptops, tablets and smartphones, anyone can find the perfect device that suits his or her needs while factoring mobility, size or processing power. ***Browser fingerprinting*** became a reality thanks to the software and hardware diversity that compose every single one of our modern devices. By collecting device-specific information with a simple script running in the browser, a server can fully or partially identify a device on the web and follow it wherever it goes. This technique presents strong privacy implications as it does not require the use of stateful identifiers like cookies that can be removed or managed by the user. In this thesis, we provide three contributions to the browser fingerprinting domain:

1- We perform **the analysis of 118,934 genuine fingerprints** from the *AmIUnique.org* website. We show that device identification is possible as 89.4% of collected fingerprints are unique. Notably, we observe that browser fingerprinting can be extended to mobile devices despite their constrained nature in terms of software and hardware.

2- We design **two countermeasures called Blink and FPRandom**. Since the information contained in a fingerprint is primarily used to tailor the browsing experience to the user's device, modifying it correctly presents great challenges as the wrong information can be detected and break the browsing session. By taking advantage of software diversity and randomness, we constantly change the values collected by tracking scripts so that they are unable to identify a device online.

3- We detail **a complete protocol based on canvas fingerprinting to augment authentication on the web**. At a time where passwords and logins are considered insecure with the numerous data breaches reported each year, we provide a new way to assert the identity of a device. By asking the browser to render very specific images in a canvas element, we can verify that a device is a known one, preventing account takeover through simple credentials theft.

Browser fingerprinting is still in its early days. As the web is in constant evolution and as browser vendors keep pushing the limits of what we can do online, the contours of this technique are continually changing. With this dissertation, we shine a light into its inner-workings and its challenges along with a new perspective on how it can reinforce account security.

INSA RENNES

N° d'ordre : **17ISAR 25 / D17 - 25**

Institut National des Sciences Appliquées de Rennes
20, Avenue des Buttes de Coësmes - CS 14315 - F-35043 Rennes Cedex
Tél : 02 23 23 82 00 – Fax : 02 23 23 83 96