Web Scrapology

Overcoming limits of automating web measurements

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Open Universiteit
op gezag van de rector magnificus
prof. dr. Th.J. Bastiaens
ten overstaan van een door het
College voor promoties ingestelde commissie
in het openbaar te verdedigen

op vrijdag 8 december 2023 te Heerlen
om 13:30 uur precies

door
Benjamin Krumnow, MSc
geboren op 25 maart 1985 te Berlijn

Promotor:
Prof. dr. ir. H.P.E. Vranken, Open Universiteit

Copromotores:
Prof. dr. S. Karsch, TH Köln
Dr. ir. H.L. Jonker, Open Universiteit

Leden beoordelingscommissie:
Prof. dr. ir. G.J.P.M. Houben, Technische Universiteit Delft
Prof. dr. S. Mauw, Université du Luxembourg
Prof. dr. T.E.J. Vos, Open Universiteit
Dr. M.G.C. Acar, Radboud Universiteit
Dr. S. Calzavara, Università Ca' Foscari Venezia

This book only uses FSC-certified paper and offsets CO_2 emissions.

Summary

Web measurements provide insights into the World Wide Web's performance, security, and privacy. Measuring a website appears to be simple: let a measurement tool interact with it and perform the measurements. To achieve scale, automate the process and set it to visit all desired sites. However, not all web users necessarily get to view the same Web. For example, online advertising typically makes use of tracking techniques, a privacy invasive measure, but advertisers tend to take measures to not show advertisements to automated visitors in order to prevent fraud. Measuring web privacy needs to account for such effects.

In general, automated measurement tools should be able to collect data from the Web as users experience it. That is, being able to go beyond the Web shown only to automated visitors. This becomes a necessity when web measurement tools are used to gain insight into online privacy and online security on the Web. To this end, measurement tooling must overcome two obstacles. First, **websites limit reachability of content**. They do so not to stop automated measurements, but for functionality reasons. Secondly, **websites intentionally oppose automated visitors**. They may use blocking mechanisms such as CAPTCHAs, or, more insidiously, tailor responses to automated visitors, e.g., leaving out advertisements or videos. From a measurement perspective, obstacles may result in blind spots which undermine a study's validity and limit its significance. This thesis explores and overcomes limitations of automated web measurements.

In the first part, we address measures that limit content reachability as a side effect. We begin with overcoming the login barrier by designing a method to enable automated login. We develop a framework, Shepherd, implementing this approach. We validate Shepherd's ability to achieve scale by performing an in-depth case study of web session security over thousands of sites. Next, we turn to measuring differences in views offered to different clients. For that, we develop a framework to enable data acquisition from multiple platforms simultaneously, enabling studies to discover differences in how data is presented to specific clients. We validate its efficacy by performing a price comparison study of flight vendors, acquiring data from these vendors' desktop and mobile shops simultaneously. Our application is successful in collecting data simultaneously, allowing us to identify strong indications of price differences between platforms.

In the second part, this thesis focuses on methods used by websites that deliberately disturb a bot's functionality and therefore undermine the reliability of automated measurements. We explore how fingerprint-based detection works and develop

an automated approach to construct a bot's fingerprint surface. We present a study of 1 million websites to determine the widespread of fingerprint-based detection. Next, we address behavioural detection by developing the first Selenium-ready framework that can simulate human interaction. Last, we explore the effects of bot detection in a case study based on OpenWPM. OpenWPM is a web privacy measurement tool. We investigate its reliability in the context of an adversary website. Our investigation shows that OpenWPM is prone to fingerprint-based detection and attacks on its data recording. We develop countermeasures that circumvent our found attacks and detection methods. Finally, we compare measurements taken with regular OpenWPM to our countermeasures-enriched version of OpenWPM. We find significant differences that can affect privacy web measurements. From this, we conclude that countermeasures are necessary for reliable measurements with OpenWPM.

Overall, this thesis shows that both types of obstacles can be overcome, albeit at the cost of considerable engineering efforts. We find significant differences arise from overcoming certain obstacles. We thus conclude that, for some types of web measurements, such efforts are necessary.

Samenvatting

Webmetingen geven inzicht in de prestaties, veiligheid en privacy van het World Wide Web. Het meten van een website lijkt eenvoudig: laat een meetinstrument ermee interacteren en voer de metingen uit. Om dit op grote schaal uit te voeren, automatiseer je het proces. Echter: niet alle webgebruikers krijgen noodzakelijkerwijs hetzelfde te zien. Zo nemen adverteerders maatregelen om geen advertenties te tonen aan geautomatiseerde bezoekers (web bots) om fraude te voorkomen. Daarbovenop gebruiken advertenties vaak trackingtechnieken, een privacyschendende maatregel. Dat levert een uitdaging op voor het meten van online privacy.

Idealiter zou een geautomatiseerd meetinstrument het Web moeten kunnen meten, zoals reguliere gebruikers het ervaren, niet alleen het Web zoals het enkel aan geautomatiseerde bezoekers wordt getoond. Dit is noodzakelijk voor webmetingen van online privacy en online security. Hiertoe moeten meetinstrumenten twee obstakels overwinnen. Ten eerste zijn er stukken van websites die niet voor iedereen toegankelijk zijn. Het tweede obstakel is dat websites somes web bots proberen buiten te houden. Dit kan door de site te blokkeren, bijvoorbeeld door middel van CAPTCHA's, maar ook door de inhoud on-the-fly aan te passen aan de bezoeker. Zo kunnen sites advertenties niet tonen aan web bots om advertentiefraude te voorkomen, of video's niet tonen om minder kosten te hebben aan verstuurde data. Hoewel er legitieme redenen zijn om zulke maatregelen te nemen, dreigen dit soort obstakels in blinde vlekken voor webmetingen te resulteren. Dat ondermijnt de validiteit van de meting. Deze dissertatie onderzoekt deze types obstakels voor geautomatiseerde webmetingen en demonstreert hoe deze overwonnen kunnen worden.

In het eerste deel behandelen we obstakels die als neveneffect de bereikbaarheid van inhoud beperken. Als eerste onderzoeken we het type obstakel waarbij de web bot niet alle inhoud van de website kan zien. Bij dit obstakel richten we ons op inhoud achter logins en op verschillen in getoonde inhoud. We overwinnen de login barrière door een methode te ontwerpen om geautomatiseerd inloggen mogelijk te maken. Dit ontwerp implementeren we in een tool, Shepherd. We valideren de schaalbaarheid van Shepherd door een diepgaande casestudy uit te voeren van de beveiliging van websessies op duizenden sites.
Vervolgens richten we ons op het meten hoe websites verschillende typen bezoekers (desktop, mobiel) verschillende inhoud tonen. Hiertoe ontwikkelen we een tool om gegevens van meerdere platforms tegelijk te verzamelen. Met behulp van dit tool kunnen onderzoekers te weten komen of er verschillen zijn in de inhoud die verschillende types bezoekers te zien krijgen. We valideren de doeltreffendheid van de tool

door een cross-platform prijsvergelijkingsstudie uit te voeren. We vergelijken prijzen van vliegreizen van verschillende vluchtaanbieders, waarbij we simultaan gegevens verkrijgen van de desktop-site, de mobiele site en de mobiele app. Onze toepassing is succesvol in het simultaan verzamelen van deze gegevens, waardoor we sterke aanwijzingen van prijsverschillen tussen deze platforms kunnen identificeren.

In het tweede deel richt dit proefschrift zich op methoden die gebruikt worden door websites die opzettelijk web bots bestrijden en hierdoor de betrouwbaarheid van geautomatiseerde metingen ondermijnen. We onderzoeken hoe web bots gedetecteerd kunnen worden op basis van kleine verschillen in de zogenaamde 'browser fingerprint' vergeleken met reguliere browsers. We ontwikkelen een geautomatiseerde aanpak om de hieruit volgende 'fingerprint-surface' van een bot te bepalen. We analyseren de Top 1M websites om te bepalen in hoeverre fingerprint-based bot detection in de praktijk gebruikt wordt.
Dit volgen we op door web bot detectie door middel van gedragsdetectie te bestuderen. We ontwikkelen een raamwerk voor de browser-automatiseringstool Selenium dat Selenium op realistisch menselijke wijze laat interacteren met webpagina's.
Tot slot onderzoeken we de effecten van botdetectie in een casestudy. OpenWPM is een meetinstrument voor webprivacy. We onderzoeken de betrouwbaarheid ervan in de context van een website die actief probeert OpenWPM's metingen te dwarsbomen. Uit ons onderzoek blijkt dat een website OpenWPM kan herkennen aan zijn browser fingerprint – en dat OpenWPM vatbaar is voor aanvallen op de dataverzamelingsfuncties. We ontwikkelen tegenmaatregelen die de gevonden aanvallen en detectiemethoden verhelpen danwel omzeilen. Tot slot vergelijken we metingen tussen de reguliere OpenWPM-versie en onze, met tegenmaatregelen verrijkte, OpenWPM-versie. We vinden significante verschillen in resulterende privacymetingen tussen de tools. Hieruit trekken we de conclusie dat tegenmaatregelen noodzakelijk zijn voor betrouwbare metingen met OpenWPM.

In het algemeen toont dit proefschrift aan dat beide soorten obstakels kunnen worden overwonnen, zij het ten koste van aanzienlijke technische inspanningen. We vinden significante verschillen door deze barrières te overwinnen. We concluderen dan ook dat dergelijke inspanningen, voor sommige typen webmetingen, noodzakelijk zijn.

Acknowledgments

First of all, I would like to express my sincere gratitude to my supervisors who have accompanied me over the years. In particular, Hugo Jonker, who made this time so special. Thank you for the incredible enthusiasm and passion that you brought to our research. You pushed me to make the best of my work and my time as a Ph.D. student. You were always honest with me, sparing neither compliments nor criticism. Thanks to you, I have matured as a researcher and as a person. I would also like to thank Marko van Eekelen and Harald Vranken, who took the role of my promoter and accepted me as a PhD student at the OU. You were always positive and supportive during this time. Thank you, Harald, for stepping into the breach and taking the lead at the end of my dissertation. I am also grateful to Stefan Karsch and Hans Ludwig Stahl for providing the environment to complete my dissertation at the TH Köln. Thank you for your trust and giving for me the opportunity to go on this journey.

I am particularly thankful to have met Gerhard Hartmann and Kristian Fischer. Both shaped me in the early stages of my academic career and laid the foundations for me to flourish. Thank you, Kristian, for all your guidance and Gerhard for being a great mentor and bringing me into the Web Science programme.

I feel fortunate that I could collaborate with some bright researchers and individuals. Our collaborations helped me grow, and I look back proudly on our work. Further, I met with many clever students during their final projects and classes. In some cases, these students made an exceptional effort during their theses, leading to publications. Therefore, many thanks go to Gabry Vlot, Marc Sleegers, David Roefs, Daniel Goßen, and Godfried Meesters.

Further, I thank the committee and reviewers that improved my work by providing comments, discussions, and criticism and helped me to become a better researcher. I would also like to thank Christopher, Bianca, Sabine, and Patrick for helping with proofreading my work and eradicating errors.

I am also thankful for my colleagues in the KTDS lab. Thank you, Ahmad, Christopher, Dominik, JJ, Linda, Sabine, and Svenja, for sticking with me through long days of proofreading, for providing the allegedly best candy buffet in town, and for the many fun journal club meetings we had together. I will miss your company at my next destination.

In addition to my professional environment, I have had tremendous support from my friends and family. Thanks for all the time we spent together and for giving me strength and joy outside of the lab. I am sure that this thesis would not exist without you. Thank you, Brigitte, Jenni, Julia, Steffen, Rosa, Rudi, Waltraud, Anne, Patrick,

Janna, Joschka, Alan, Dean, Marcel, Matthias, Christian, Thorsten, Florentine, and the people at Sportschule Shintai, especially Klaus, who kept me in shape.

Special thanks for their mental support go to Joschka and Patrick. Thank you for more than a decade of loyal and deep friendship, despite the hundreds of kilometres between us. I would also like to thank my mother and my sisters. With your love, comfort, and care, you give me the feeling of being at home. I am lucky to have you.

Finally, I am very grateful to have two such wonderful souls around me. Thank you, Ellie, for all the joy you have brought into our lives and for making me laugh every day. If ever I felt there was more to life, you were it. Thank you Kathi, my partner, love, and soul mate. Your incredible encouragement, endless care and undying belief in me has made this work possible. After every setback you built me up and gave me the strength to carry on. Sharing this success with you is the greatest joy of all. Thank you for your wonderful love and the amazing time I get to spend with you.

Contents

Chapter 1

Introduction

Web measurements are instrumental to a secure Web. They provide insights into how part of the Web is used (or misused), bring privacy violations to light, and report on the effectiveness and proliferation of security measures. The scale of a web measurement study often defines the scope of gained insights; wherefore, it is is an essential requirement. A standard tool to achieve scale in measurements is web automation, i.e., scripting web clients to autonomously interact with web servers. Ahmad et al. [ADZ+20] reported that 16% of studies published at major security and privacy venues nowadays use automated clients (also called bots or web bots). While various frameworks (e.g., Scrapy, Selenium, Puppeteer, etc.) exist that automate web client interaction, these lack functionality to record data of such interaction, such as measurements on cookie operations, HTTP messages, and requested DNS records. In response, several measurement frameworks have been developed, e.g., OpenWPM [EN16], VisibleV8 [JK19], Tracker Radar Collector [Duc23], and others. These combine client automation and measurement instrumentation out of the box. Making instrumentation accessible reduces some heavy lifting researchers face when conducting web measurements. Still, an important issue is that the use of tooling must not skew the outcome of the experiment. However, as with most technology, web automation can have side effects posing obstacles to measurements. For example, advertising networks pay for ads to get the user's attention. When they show ads to a bot, they spend money for no effect. So, it is in an advertiser's interest to treat bots differently from human visitors, e.g., by not showing any ads. Such practices do not only limit the significance of an automatically conducted measurement, operators can use them to hide their real doings. Consequently, drawing conclusions from measurements that do not account for such obstacles inherit the risk of developing incomplete and flawed privacy and security solutions for the Web.

This thesis aims to overcome obstacles for web measurement studies. Hence, it must investigate those obstacles to devise appropriate methods and tooling to achieve this goal. However, overcoming obstacles may impact web measurements, which must also be regarded. To cover both aspects, we formulate the following main research question for this thesis:

> *Main RQ: How to overcome obstacles for automated web measurements and what is the impact of doing so?*

In this work, we recognise two main obstacles that web measurements face: those that personalise and tailor the content behind them to the visitor and those that

keep out automated visitors. In more detail, some standard features and behaviours on websites offer desirable functionality, but, as a side effect, pose difficulty for web measurements. Since their primary purpose is different from interfering with web measurements, nor is it to stop automated visitors, we call such features *unintended obstacles*. In contrast, obstacles that treat bots differently from human users serve a different purpose. Such bot defences are put in place to protect a party (e.g., a site or script) against automated threats or to mislead automated visitors, wherefore we name them *deliberate obstacles*.

In general, we see two typical situations where unintended obstacles occur: In the first situation, a website limits the reach of an automated visitor as it makes its pages or elements difficult to find or access. A common challenge is website areas (e.g., specific pages, elements, or functionality) that require an authenticated session. Given that more than half of the Top 100K websites possess a login page [vAHS17], automated visitors cannot access these areas on a significant portion of the Web. The second scenario concerns tailored website responses based on different criteria, such as online price differences based on a client's history [HSL+14] and fingerprint [HTW+18], different website versions due to a client's device class [vGPJ19; YY20], or deviating content based on a user's geolocation [FMS+15; ITK+16; EAW+19]. Consequently, relying on one user agent to explore a website may reveal only some variants of a site.

In contrast, deliberate obstacles occur when websites use bot defences to react on bot visitors. Bot defence measures fall into two groups: those that rely on identifying the visitor as a bot and those that do not need this. For example, measures such as blocking CAPTCHAs and randomising elements to prevent scraping may be passive, that is, they can be applied to all visitors. Other measures rely on detection; for example, not wasting bandwidth on bots (by not delivering videos or images) requires classifying the visitor as a bot. While preventive measures stop the interaction with a bot, bot detection is more subtle, enabling a party to pick any suitable countermeasure, including CAPTCHAs. Hence, our focus lies on bot detection methods.

Putting these two different purposes for obstacles for automated web measurements into the context of the main research question, we arrive at two sub-research questions for each purpose.

RQ1.1: How to overcome unintended obstacles?
RQ1.2: What is the impact of overcoming unintended obstacles?

RQ2.1: How to overcome deliberate obstacles?
RQ2.2: What is the impact of overcoming deliberate obstacles?

Nevertheless, RQ1.2 and RQ2.2 are difficult to assess in a generic fashion. Web measurements are diverse and the impact on measurements varies with the goal and parameters of a specific study. Thus, any attempt to answer such question will be incomplete. Still, we can approach the answer in the context of this thesis by evaluating conducted research.

1.1 Thesis contributions

The research in this thesis offers three types of contributions: experiments that measure aspects of the Web, methods to acquire and analyse such data, and investiga-

tions to challenge the security of automated web measurement tools. Due to the ever-evolving nature of the Web, repeating the experiments will result in different measurements. Similarly, technological advancements might necessitate the expansion of the acquisition and analysis methodologies. However, the concepts behind these methodologies do stand the test of time. With respect to the research questions above, this thesis makes the following contributions.

Key contributions in overcoming unintended obstacles. This thesis contributes by designing and implementing tools that increase web measurement coverage. We identify two open research areas relating to tailored content where this thesis provides new tools and insights.

- *Design and implementation of a framework to automatically log in on websites.* User authentication is a standard procedure to provide access to personal data and sensitive functionality online. We explore challenges for automation frameworks to reach post-login areas while treating the suspected website as a black box. We build a framework to enable end-to-end automation of the login process. We show that studies covering thousands of sites with logins are possible while also accomplishing a better success rate in reaching post-login areas per attempt with respect to previous work. Till today, our framework has been used in three peer-reviewed publications [MADWeb20; CoSe21; SecWeb21].

- *Methodology and framework to elicit view-dependent price differentiation.* Websites adjust content to visitors based on various factors. Deducing the influence of a single factor can be difficult from an outsider's perspective. This thesis systematically studies price differentiation depending on the chosen distribution channel (also called view). In detail, we devise a method and develop a framework to collect data from multiple views. We evaluate our framework by conducting a price-comparison study of travel agents using mobile phones, desktop browsers, and different localisations, showing that price differences between views exist.

Key contributions in overcoming deliberate obstacles. To overcome deliberate obstacles, we study how websites can recognise that a visitor is a bot directly or indirectly. To directly identify a bot, a party must know beforehand what distinguishes this specific bot. Hence, this approach can only detect instances of specific automated visitors. In contrast, a party can gather data that reveals patterns diverting from human visitors. While this approach can detect previously unknown bots, it requires more time and a sufficient number of interactions by the client. We make the following contributions to overcome both methods.

- *Methodology and tooling to construct a bot's fingerprint surface.* A requirement for specific detection is that a bot inhibits properties distinctive from clients controlled by human visitors. The ability to reliably identify such properties enables both the development of detection and anti-detection techniques. Our contribution is a generic method to construct a bot's fingerprint surface – a set of distinctive properties. We evaluate our approach against famous automation and measurement frameworks, revealing identifiable properties in most of them.

- *Measurements to determine the proliferation of direct bot detection.* Based on our findings above, we devise a method to evaluate whether a website includes bot detection scripts. Using this method, we conduct a scan on a million websites. Our study suggests that at least 12% of websites deploy bot detection.

- *Model, method and implementation for human-like interaction simulation.* A client's interaction characteristics may reveal whether a client is a bot. However, to what extent automation frameworks account for this aspect or whether a solution exists to simulate realistic interaction by web users is not well-known. We investigate Selenium's interaction characteristics, one of the most popular automation frameworks, and provide a library to simulate human-like behaviour. Finally, we propose a theoretical model that defines certain levels of realism, allowing developers to classify detectors and simulators by their capabilities.

Key contributions in measuring the impact of overcoming obstacles. To address this aspect, we provide two case studies that require overcoming unintended or deliberate obstacles. The two related chapters are marked with case study in the title. The case studies' contributions can be summarised as follow:

- *Large-scale evaluation of session security flaws in the wild.* Many previous studies have tested for session security flaws. However, none of these considered logging in, reached a large number of websites, or included a comprehensive security evaluation. We present the first study that provides an extensive web session security evaluation while also reaching a significant number of websites. To accomplish an extensive security evaluation, we collect data from all phases of a user session: pre-login, logging in, post-login, and after logging out while testing for a wide number of known flaws for web sessions.

- *Hardening of a measurement framework and measuring the effect of hardening.* Given that 12% of sites deploy bot detection, it becomes an important issue how bot detection influences web measurements. However, as long as a measurement framework's fingerprint surface and instrumentation reliability is unknown, this issue remains challenging to address. To that end, we provide a deep investigation of the resilience of a well-established web privacy measurement framework against detection and attacks on its instrumentation. We develop an improved version that resists fingerprint-based detection, and all found attacks. Finally, we determine the differences between a hardened and a non-hardened version, and find that privacy studies can be significantly affected.

1.2 Thesis outline and author contribution

The contributions of this thesis spread over two main parts. The first part "*Increasing Measurement Coverage*" addresses challenges relating to the unintended obstacles for web measurement studies. The second part "*Advancing Attacks and Defences*" deals with deliberate obstacles and approaches to circumvent such obstacles. As this thesis follows a cumulative publication model, each chapter in the main parts relies on one peer-reviewed publication. An overview of these chapters, their underlying

publications, and a list of the author's contributions to these publications is depicted in Figure 1.1. The following paragraphs provide summaries for each chapter and describe the author's contributions. Note that well-defined terminology avoids the problem of an ambiguous interpretation. Therefore, we use the Contributor Roles Taxonomy (CRediT) [BAA+15] which provides such terminology.[1]

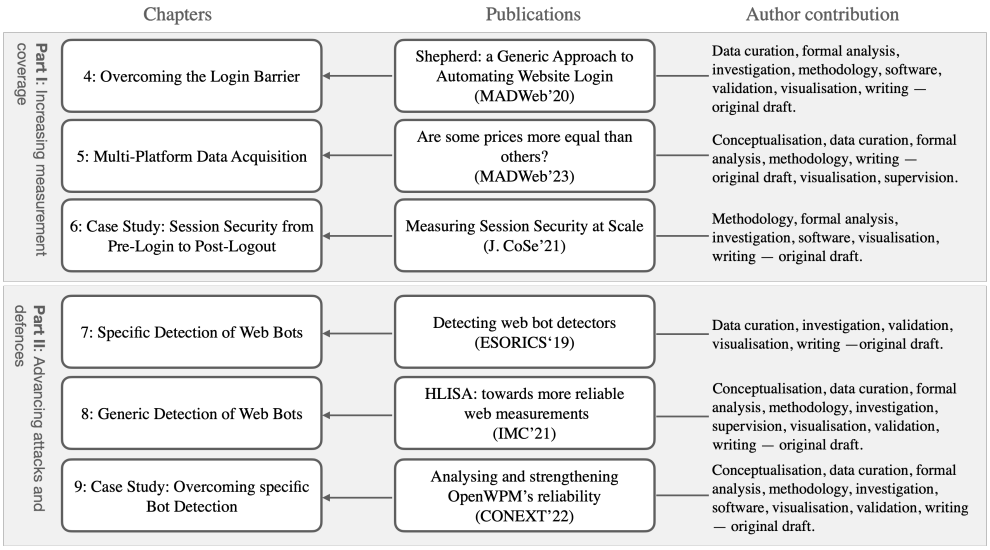

Figure 1.1: At a glance: thesis structure, publications, and author contributions

Part I – Chapter 4: Overcoming the Login Barrier. This chapter introduces Shepherd, a framework to automate the login process on websites. To that end, Shepherd offers automated routines for sourcing credentials, finding login pages, conducting login routines, validating login success, and logging out. In an initial evaluation, Shepherd logged in on over 7K websites of a set of ∼50K websites. The chapter concludes with a performance comparison between Shepherd and previous approaches as well as a discussion of potential use cases.

Personal contribution. Marc Sleegers originally invented Shepherd during his bachelor thesis [Sle17]. He wrote the first version of Shepherd that processes static HTML pages. I re-wrote Shepherd from scratch to make it fully compatible with modern websites that serve dynamic content. Together with Hugo Jonker, I am the primary author of the resulting publication [MADWeb20]. I carried out the investigation, formal analysis, validation and created visualisations.

Part I – Chapter 5: Multi-platform Data Acquisition. Chapter 5 proposes a data collection method to simultaneously collect prices from different platforms. It presents an implementation that manages synchronous data collection of different sites and heterogeneous clients, such as mobile devices and desktop computers. Finally,

[1] A brief explanation of terms defined in CRediT can be found on `https://www.elsevier.com/authors/policies-and-guidelines/credit-author-statement`

it reports on our evaluation of five travel vendors, where we find vital signs that platform-specific pricing exists in practice.

Personal contribution. I initiated the concept of this project and outlined it into a master's project. With Hugo Jonker, I supervised Godfried Meesters' master project [Mee21], intending to publish the results. For the publication, I fully reworked the data analysis. Hugo Jonker and I authored the resulting paper [MADWeb23].

Part I – Chapter 6: Case Study: Session Security from Pre-Login to Post-Logout. This chapter shows a specific application of Shepherd – a large-scale and comprehensive session security study on websites. It starts with describing how we created a post-login-logout data set and investigates its representativeness. Then, it dives into the security assessment of ∼6K websites, which provides insights into all phases in the lifecycle of an authenticated user session for these sites. The evaluation demonstrates the prevalence of well-known session security flaws.

Personal contribution. The journal paper [CoSe21] used in this chapter is a joint effort by all authors. I contributed to the methodology, enhanced Shepherd, performed the investigation and participated in the data analysis. Lastly, I wrote a significant part of the paper.

Part II – Chapter 7: Specific Detection of Web Bots. This chapter shines a light on browser fingerprinting techniques that distinguish automated from regular web users. It starts with investigating the inner functioning of fingerprint-based bot detection by reverse engineering a commercial bot detector script. Insights from this process led to a methodology for determining unique properties in automation frameworks. This chapter continues with investigating state-of-the-art automation frameworks to reveal their identifiable properties. It further assesses their use for bot detection by conducting a 1M sites web measurement study. Finally, this chapter provides insights into bot detection's effect by creating a less detectable automated Chrome browser to visit sites with detectors.

Personal contribution. The initial idea for this project arose during our work on Shepherd. Gabry Vlot took up this project under the supervision of Hugo Jonker and Greg Alpár. He investigated the commercial bot detector, developed the used tools and conducted an initial measurement in his master's thesis [Vlo18]. Hugo Jonker and I authored the resulting conference publication [ESORICS19]. For this publication, I re-executed the investigation, performed the formal analysis, and designed and conducted the validation.

Part II – Chapter 8: Generic Detection of Web Bots. Fingerprint-based bot detection is limited to the known set of identifiable properties in automation tools. Generic detection approaches can overcome this limitation. This chapter investigates whether the famous automation framework Selenium is prone to detection based on artefacts in its behaviour. Second, it proposes HLISA, a framework to add human-alike interaction to Selenium-based bots. It then introduces a theoretical framework which models the arms race between detectors and simulators regarding their capabilities. Finally, according to our proposed framework, it ranks HLISA's capabilities to circumvent bot detection.

Personal contribution. The underlying paper [IMC21] relies on two student projects. Daniel Goßen [Goß20] investigated modifications to the OpenWPM framework to hide identifiable properties, and David Roefs [Roe21] developed a framework to mimic human-like interaction. The ideas for both projects arose during my work on Chapter 9. I co-supervised David's project and supported Daniel with my findings at the start of his project. Hugo Jonker and I are the principal authors of this paper. I conducted the validation and formal analysis and contributed with visualisations.

Part II – Chapter 9: Case Study: Overcoming specific Bot Detection. This chapter evaluates the reliability of measurement frameworks in the presence of hostile websites. To that end, it provides a case study of OpenWPM – an often-used web privacy measurement framework. The results of the presented investigation uncover identifiable properties and attacks against the reliability of this framework. Then, this chapter provides a re-implementation of OpenWPM's instrumentation, which is resistant to the previous found attacks and less detectable. Experiments with this new version show a significant difference from vanilla OpenWPM with substantial effects on privacy measurement studies.

Personal contribution. The concept for this work arose from the publication used for Chapter 7 and in discussions with my supervisor Hugo Jonker. I took the lead in planning, designing, and performing the research, as well as authoring and presenting the paper [CoNEXT22].

Chapter 2

Background

This thesis examines issues concerning the automation of web measurements. We assume that the reader is familiar with the fundamental technological concepts of the Web; hence, we focus on the relevant materials for automated web clients. We start with building blocks for automating web measurements (Section 2.1). Then we derive a classification based on some of the building blocks and existing web automation approaches (Section 2.2). Afterwards, we look at the ethical aspects of automation and how we address common issues to reduce potential harm (Section 2.3).

2.1 Principles of web measurement automation

The Web is an information space in which machines operate by exchanging messages amongst each other [BBC+04]. In the context of this thesis, the primarily encountered machines operate according to the traditional client-server model. A web study must collect and analyse data on the interaction between these parties to examine the Web. There are two approaches to collecting such data. First, a study can collect data from web users during their interaction with servers, e.g., through soaking up network traffic or with the help of crowd-sourcing.[1] Second, it can run its own web clients. The results vary depending on various factors, such as the observed client characteristics and targeted websites. A study cannot control all of these factors. However, in the latter approach, the study must define the clients and targeted sites. This raises the point of how it should set these parameters. In this section, we describe how studies select websites and three aspects that may be crucial for determining suitable web clients: the internal subsystems used to process modern websites, capabilities for executing code to support dynamic websites, and techniques to re-identify web clients.

2.1.1 How to choose websites for studying

Any web measurement study needs to define its target domains. A desired goal in many studies is to conduct the measurement on sites that approximate the experience of the majority of web users. That is, the more popular a site, the better the approximation. Top lists that rank websites by popularity may serve this purpose best. However, the determination of popularity depends on the particular top list

[1] i.e., transferring tasks to volunteers or/and paid individuals

creator. For example, top lists like the Alexa Global Top 1M, the Cisco Umbrella 1M, the Majestic Million, and Quantcast have gained some popularity in research. Still, each of these lists relies on individual metrics for constructing its ranking.[2] Previous work [SHG+18; LVT+19] has questioned whether these lists fulfil requirements of scientific research, such as representativeness, soundness and reproducibility. A result of this process is the Tranco list,[3] a more robust list as it combines multiple top lists to achieve better domain stability. It should therefore be the preferred solution for researchers over top list without measures to ensure robustness.

2.1.2 Bot terminology

The literature uses many terms to refer to clients who have been automated, but most of them imply a specific application context. For example, previous work (e.g., [Eic95; TS06]) has used the terms crawler and spider to denote automated clients indexing web pages or content. In more detail, such clients traverse pages of one or multiple websites to search for links to other pages or to download certain documents. Similarly, a scraper primarily aims to extract website content (e.g., [GLL+14]). Typical examples of scrapers are web clients that automatically collect images or pricing information from websites. Throughout this work, we prefer the term web bot, as to the best of our knowledge, it does not imply a limited scope. We define a web bot, with respect to the definition in [Mer22], as a programme that automates tasks using web technologies. We will use this term interchangeably with automated visitor or client.

2.1.3 Browser subsystems

Web browsers perform similar duties to display and interact with websites. However, there is no common standard to develop a browser [Kos18], wherefore browsers will vary in their implementations. Still, it is possible to abstract from specific implementations to identify reoccurring components across multiple browsers. Grosskurth and Godfrey [GG05] took up this idea and proposed a reference architecture that describes a browser's subsystems and their dependencies. Their model consists of eight subsystems (depicted in Figure 2.1). As their work dates back to 2005, their model may not account for systems providing newer functionality (e.g., WebAssembly[4]). However, their model is sufficient to distinguish between different automated web clients, which we use in Section 2.2. We summarise these subsystems as follows:

- **User interface**: contains interactive elements for user input and passes on user input to the browser engine.
- **Browser engine:** gives control over the rendering engine and can alter its behaviour. Some browser variants implement the browser and rendering engine in the same subsystem. A rendering engine creates graphical representations of the loaded content. To do so, it exchanges information with multiple other subsystems. In the remainder of this thesis, we will discuss them as one.

[2] For further details on the composition of these lists, we refer to the work by Scheitle [SHG+18] and Le Pochat et al. [LVT+19], that recently reported on the metrics used in these lists.

[3] `https://tranco-list.eu/`

[4] `https://developer.mozilla.org/en-US/docs/WebAssembly`

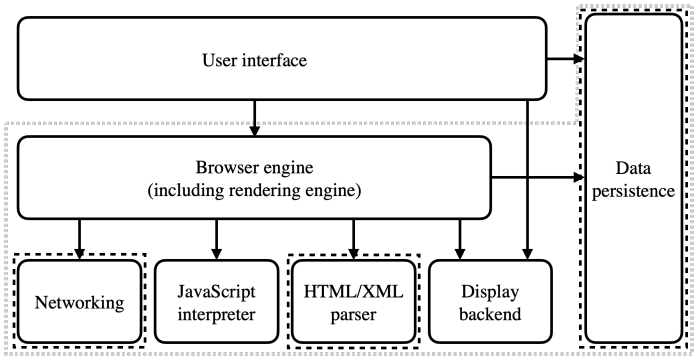

Figure 2.1: A reference architecture for browsers (adopted from [GG05]).

- **Networking subsystem:** takes over communication-related tasks, such as requesting documents via HTTP.
- **JavaScript interpreter/engine:** runs web page scripts or browser extension scripts.
- **HTML/XML parser:** parses HTML/XML documents to build the Document Object Model. Note that some (e.g., [GI11]) see this subsystem as a part of the rendering engine.
- **Display backend:** provides primitives for graphical representations. The primitives can depend on the underlying OS.
- **Data persistence:** stores website data, session data, user settings, and so on.

Although plenty of browsers exist today, most web users see the Web through not more than three browser engines.[5] These engines are Blink (Chrome and Chromium variants like Edge, Opera, Samsung Internet, etc.), Webkit (Safari and iOS-based browsers), and Gecko (Firefox). All three engines rely on different JavaScript engines: SpiderMonkey (Gecko), V8 (Blink), and JavaScriptCore (Webkit). However, browser-based JavaScript engines also exist outside of the web browser environment. For example, Node.js uses Chrome's V8 JavaScript engine [CFA+23].

2.1.4 The browser execution environment

Browsers construct the Document Object Model (DOM) to enable dynamic web pages. The DOM is an object-based representation of an HTML document in memory. Figure 2.2 depicts a reduced DOM in a browser environment for the HTML code in Listing. 2.1. Notably, the objects follow a tree hierarchy with the *window* object as the root. The window object represents the current document in a browser tab. The parsed version of a web page's HTML content is located under the document object. Besides the document object, several other objects (e.g., navigator, storage, etc.) descend from the window object. These objects, including the window object, are

[5] Given browser market share statistics from 2023 [Sta23; Sim23]

```
<!DOCTYPE html>
<html>
<head>
<title>Web Scrapology</title>
</head>
<body>
<div>
<h1>Example HTML document</h1>
<p>Lorem ipsum dolor sit amet,
    consectetur adipiscing elit.
</p>
</div>
</body>
</html>
```

Listing 2.1: Example HTML document

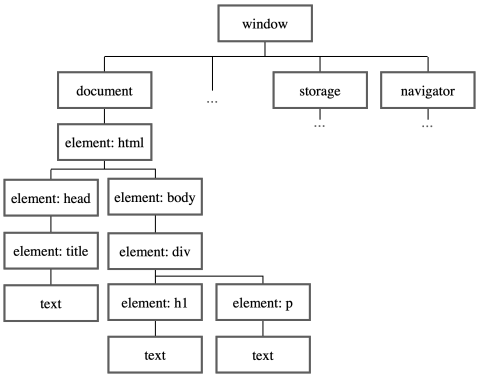

Figure 2.2: Corresponding DOM tree in a browser environment

interfaces for advanced interaction with the browser. They cover a broad functional spectrum for developers, such as network communication, storing data persistently, retrieving information from the browser, and playing audio and video.

Websites run JavaScript or CSS inside the browser to alter the DOM and interact with the browser. For that, browser vendors follow standards so that websites can remain functional regardless of the used browser platform. The ECMA TC39 committee specifies the JavaScript API and its expected behaviour in the ECMAScript standard. The latest approved version is ECMA-2622,[6] which is available in all modern browsers. However, the ECMAScript is independent of interfaces defining the interaction with web pages and the browser environment, as these are specified in separate standards. The document object is the interface to operate on the DOM and implements the DOM specification. This specification (earlier called DOM levels) is nowadays a living standard, that is, a specification that can receive further non-breaking updates, even after its official release. In contrast, the Web API[7] defines interfaces to interact with the browser. Organisations like the Web Hypertext Application Technology Working Group (WHATWG) and the W3C take care of the standardisation.[8] In addition, they define the Web Interface Definition Language[9] (Web IDL), the binding between Web API and JavaScript.

Note that a web client may divert from the standards above, which is true even for mainstream browsers. For example, Apple declined to implement specific Web APIs that pose a risk to web user privacy [Cim20], while Chrome implements these features. A user may also use a preliminary browser, whose version contains features that are not yet part of a stable release. Other reasons may be browser extensions or frameworks that alter the DOM. These must not change standardised functionality but can make a client more distinguishable when changing DOM properties [KIS+20].

[6] `https://tc39.es/ecma262/`

[7] An overview as well as specific descriptions are provided in the MDN documentation under `https://developer.mozilla.org/en-US/docs/Web/API`

[8] For an overview, see `https://www.w3.org/TR/?tag=webapi`

[9] `https://webidl.spec.whatwg.org/` and `https://developer.mozilla.org/en-US/docs/Glossary/WebIDL`

2.1.5 Web client re-identification

The re-identification of web clients has many applications on the Web. Its usage encompasses the establishment of user-authenticated sessions [CoSe21], tracking of web users [RKW12; AEE+14; EN16], and the identification of web bot frameworks [ESORICS19; JK19]. Approaches for re-identification are either stateful or stateless, depending on whether they store information on the client.

Stateful re-identification. Stateful techniques identify a client by storing unique identifiers on that client. Cookies are the traditional means to store client-side information via an HTTP request or JavaScript API call. A cookie has a limited size (4KB) and is uniquely identified by a domain, URL path, and a key. Alternatively, browsers provide JavaScript-based APIs, which come with larger space. The Web Storage API offers two interfaces: `localStorage` and `sessionStorage`. Both store key-value pairs, but data in the `sessionStorage` lasts only as long as a user's tab is open. Another variant is `IndexDB`, an interface for storing large amounts of data organised in a database.

Access to stored objects underlies the Same-Origin Policy (SOP). SOP restricts access by parties (web servers and scripts) with a different origin and is enforced by the browser. A party's origin must match the stored object's protocol, host, and port, while the path can vary (cf., Figure 2.3) to satisfy the SOP. Thus, a site (e.g., `http://a.com`) setting an identifier on a visitor's client can read the identifier only from clients that visit pages belonging to the same host. This situation is commonly denoted as *first-party* context. When a.com includes a different party in the same context (e.g., via a `script`-tag), then this party (`http://b.com`) runs in the same context and may access a.com's stored objects via JavaScript. To separate both parties, a.com can also include b.com via an (e.g., via an `iframe`-tag), which makes b.com a so-called third party. Due to the third-party context, b.com manages its separate storage on a client. Moreover, it can access its stored objects on that client from any site which includes b.com in a third-party context. This access provides b.com with the ability to re-identify clients across sites (also known as third-party tracking). Note that some browsers (e.g., Total Cookie Protection[10]in Mozilla's Firefox browser) have started to isolate website caches to stop such techniques from further being used for cross-site tracking.

Figure 2.3: Example URL scheme

Stateless re-identification. In the literature, stateless re-identification is known as browser fingerprinting or device fingerprinting. Throughout this thesis, we will use the term browser fingerprinting (or just fingerprinting).

Browser fingerprinting builds unique identifiers from client properties. The premise is that clients leak information about their distinct state or composition in used hard-

[10] `https://blog.mozilla.org/security/2021/02/23/total-cookie-protection/`

ware and software components. Hence, browser fingerprinting does not require storing identifiers on the client side since the identifiable information is retrieved from the client. An exemplary workflow to create a fingerprint is as follows. During a client-server interaction, the server may examine the client's traffic characteristics which enable fingerprinting, e.g., as it may leak information about the client's used software. The server may also serve HTML, CSS, or JavaScript files, which it uses to obtain identifiable properties. Afterwards, served scripts send collected properties back to the server, which may be combined into a hash. The next time a client visits the server, it repeats the same procedure and compares the resulting hash with previously collected hashes to re-identify the client.

However, not all properties may be suitable for building a fingerprint. A fingerprint technique's usefulness depends on the *uniqueness* and *stability* of the output. When a fingerprint is not unique, a web server or script identifies two or more clients as one. The information entropy of a fingerprint determines how many clients can be separated. Even techniques with low entropy have value for fingerprinting, as combining multiple techniques increases the overall information entropy. In contrast, stability refers to the consistency of properties over time. Any change in a fingerprint's property results in a new hash that does not match the previous one. In addition, Mowery and Shacham [MS12] name three criteria that can be important for fingerprinting techniques, i.e., they should be irrecognisable for users, orthogonal to other fingerprint techniques, and easy to obtain.

2.2 Classifying web bots

Web bots come in various shapes. Some use regular web browsers to automate tasks; others resemble regular web browsers by reimplementing their functionality. In either case, their need for scalability differs significantly from those of ordinary web users. Consequently, some automated clients use custom implementations to improve stability and performance, sometimes at the cost of functionality. Such deviations can lead to different results (e.g., due to compatibility issues). This section shines a light on various forms of web bots.

We consider three entities that any bot shares to construct our classification:

1. **user commands and scripts** provide the bot's logic;
2. **automation components** enable (remote) control of web clients; and
3. **web clients** execute retrieved instructions.

Bot operators programme bots via commands or scripts that contain sequences of instructions to be executed. These can be as simple as a single command to download a web page or as complex as mimicking real user interaction. The automation component defines how user scripts must express instructions by offering an Application Programming Interface (API). It further translates instructions into specific calls that it channels to controlled web clients. Web clients are the interface to the Web or local services. Thus, they act as mediators, which substantially influence the outcome of interactions with the service and they can vary in their capabilities. While user-provided commands depend on the bot operator, the other two entities

define the technical approach to automate a web client. We use these two entities to construct our classification, as listed in Table 2.1. In the following, we describe each class and explain some selected implementations in more detail.

Table 2.1: Overview of different web client automation approaches

category	examples
1. Non-consumer clients	
(a) HTTP engine interfaces	Wget, curl, request, scrapy[11]
(b) Headless browsers	Headless Firefox, headless Chrome
(c) Custom headless browsers	PhantomJS, Nightmare, Scrapy + Splash, HTMLUnit[12]
2. Browser remote controls	
(a) Browser-integrated APIs	CDP, Marionette, browser CLIs
(b) Ready-made automation frameworks	Selenium, Puppeteer, Playwright, Cypress, etc.
3. Out-of-band controls	
– Tweak existing browser	Playwright
– UI automation	pyautogui[13], Power Automate[14], cliclick[15], etc.

2.2.1 Automation via non-consumer clients

To this category belong web bots that cover only a subset of the entire functional spectrum of a regular web browser. Thus, they need fewer subsystems than a web browser, which can give them better speed and makes them consume fewer resources. The dotted lines in Figure 2.1 show the subsystems they may include compared to regular browsers. In the following, we distinguish between three types of automated non-consumer clients.

HTTP engine interfaces are standard tools like Wget[16] and curl[17], libraries like Python's request package[18], or more advanced scraping frameworks like Scrapy.[19] Their primary functionality is to enable the scripting of networking tasks. Thus, they necessarily contain a networking subsystem to download websites but do not render a page's DOM. As a result, HTTP engine interfaces are the fastest and most suitable when the goal is to download static content from the web. They can also include or are used together with parsers to enable the selection of elements in documents and cookie handling (see black dotted boxes in Figure 2.1).

Headless browsers are modified variants of regular consumer browsers often shipped together with the official consumer browser. Headless browsers are particularly designed for web automation tasks. As their name tells, they do not offer a UI subsystem

[11] `https://github.com/scrapy-plugins/scrapy-splash`
[12] `https://htmlunit.sourceforge.io/`
[13] `https://pyautogui.readthedocs.io/en/latest/`
[14] `https://learn.microsoft.com/en-us/training/modules/pad-mouse-keyboard/`
[15] `https://github.com/BlueM/cliclick`
[16] `https://www.gnu.org/software/wget/`
[17] `https://curl.se/`
[18] `https://pypi.org/project/requests/`
[19] `https://docs.scrapy.org/en/latest/topics/architecture.html`

(grey dotted boxes in Figure 2.1) and may deviate in their capabilities and implementation from their native counterparts. For example, the standard headless variant of Chromium cannot run browser extensions [Sto17]. Since February 2023, Google has offered a version that avoids this limitation using a code base shared between native and headless Chromium [BK23]. Headless Firefox and headless Chrome/Chromium are the two common headless variants, as Apple does not provide a headless browser.

Custom headless browsers lack, similar to headless browsers, a graphical UI. In addition, they use a custom implementation of the browser engine and perhaps other subsystems. While customisation can result in better performance, it becomes more challenging to maintain the code base.[20] Custom headless and headless browsers can produce graphical representations of a webpage by rendering an image. Well-known examples of this class are PhantomJS[21] and Nightmare.[22] Nevertheless, HTTP engines can also fall under this category, but only if they are enhanced with rendering capabilities. For example, some scraping providers use Splash[23] to enable page rendering with Scrapy.

2.2.2 Browser remote controls

This category includes frameworks and interfaces that enable remote controlling for web and headless browsers. We distinguish between *browser-integrated APIs* and *ready-made automation frameworks*. The former are APIs shipped with a browser and specifically developed for the corresponding browser engine. As a result, they provide a wide range of control features, such as advanced debugging capabilities. On the other hand, this platform dependency limits their use for cross-platform automation. Variants of browser-integrated APIs are Chrome DevTools Protocol (CDP)[24] for Chrome and Chrome-based variants,[25] as well as Marionette[26]for Firefox. Besides these APIs, browsers usually allow remote controlling via a CLI.[27]

Ready-made automation frameworks enable control over multiple browser platforms. In practice, they may not reach as deep as browser-integrated APIs when giving control over the browser; for example, they may not provide access to the browser's internal debugging interface. Selenium[28] is the oldest framework in this category and the most popular framework used for web studies [ADZ+20]. It is also the principal automation framework used in this thesis. Figure 2.4 shows the components used in a Selenium-specific automation pipeline. Selenium provides high-level language-specific APIs for sending commands and retrieving responses via a (c) WebDriver client. The WebDriver client communicates via a (d) well-defined interface[29]

[20] A well-known example is PhantomJS, which got suspended due to the lack of maintainers. The original discussion can be found on its Github repository: `https://github.com/ariya/phantomjs/issues/14548`.

[21] `https://phantomjs.org/`

[22] `https://github.com/segmentio/nightmare`

[23] `https://github.com/scrapy-plugins/scrapy-splash`

[24] `https://chromedevtools.github.io/devtools-protocol/`

[25] Opera, Brave or Microsoft Edge

[26] `https://firefox-source-docs.mozilla.org/testing/marionette/Intro.html`

[27] CLI commands for Chrome can be found on `https://peter.sh/experiments/chromium-command-line-switches/`.

[28] `https://www.selenium.dev/`

[29] `https://www.w3.org/TR/webdriver/`

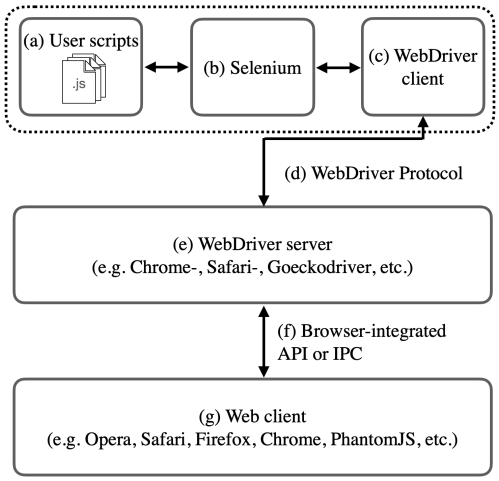

Figure 2.4: Automation pipeline using Selenium/WebDriver (modified from [Jia21])

with (e) a WebDriver Server, a browser-specific implementation. The server translates the retrieved commands to (f) browser-integrated API calls. An exception is Safari, where the driver uses inter-process communication via XPC to communicate with the browser [Jia21].

Other frameworks in this category vary conceptually from Selenium and each other. As WebDriver follows a synchronous command and response model, it does not address parallel requests well [FGS20]. This makes dealing with the event-based nature of websites more complex. In contrast, CDP uses bi-directional communication to react seamlessly to events happening in the browser. To give a basic example, a bot clicking through a site may occasionally trigger an *alert*.[30]

This event blocks all interaction on a page until closing the alert. For a Selenium-based bot, this quickly resolves into an error, as the occurrence of an alert must specifically be checked. In bi-directional communication, the automation software can listen to alert events and may react immediately. Selenium's recent releases, Selenium 4.0.0[31] and newer, also integrate a bi-directional API to handle these cases better. A framework following the bi-directional approach is Puppeteer.[32] It is built upon CDP, which binds it to Chrome-based browsers. Though, browser Mozilla is also working towards supporting CDP in Firefox [FGS21]. Some frameworks have taken a different approach to support cross-platform automation. Playwright ships its own customised browsers [Fel20] to enhance capabilities for Firefox or to simulate Webkit-based clients[33] like Safari. Yet again, Cypress[34] takes another approach. It runs inside the browser's event loop to gain enhanced control. However, according to the maintainers [Cyp22], this approach has its limitations and is, by design, unsuitable for web scraping tasks.

[30] `https://developer.mozilla.org/en-US/docs/Web/API/Window/alert`
[31] `https://www.selenium.dev/blog/2021/announcing-selenium-4/`
[32] `https://developer.chrome.com/docs/puppeteer/overview/`
[33] `https://webkit.org/`
[34] `https://www.cypress.io`

2.2.3 Out-of-band controls

Lastly, we consider automation via means outside of web browser developers' scope. First, bot operators can alter a browser's source code to provide new or additional means of control. As described above, Playwright uses this approach to automate Firefox browsers. Hence, there may be more automation frameworks in the future that provide their very own browser builds. Second, automation may happen on higher levels. To that end, UI automation software or accessibility features can take control of a browser. Here, the remote control resides in the operation system, not the browser's scope. Hence, available libraries may be platform-specific but are independent of the used browser, as long as they provide a usable interface.

2.3 Ethics of web measurement studies

The research in this thesis contributes to web bot development and web bot defences. Research in each area can potentially harm others, which merits an ethical discussion on measures to mitigate potential damage. Discussing such measures works best in the context of a particular study, wherefore we leave those discussions for the individual chapters. Instead, this section provides a broader view of ethics on web measurements. In the remainder of this section, we explore potential damage from deploying bots and analyse best practices to avoid such damage in this thesis.

2.3.1 Bot-based harm

Ethical frameworks allow us to assess harm and reason about moral decisions. Thelwall and Stuart [TS06] proposed a framework that categorises harm by bots for website providers and society. Their framework contains four categories: denial-of-service, cost, privacy, and copyright. However, nearly two decades have passed, and some aspects have gained more relevance. Therefore, we expand these thoughts with two additional categories: legislation and monetisation. We describe each category in the text below.

Denial-of-Service. Bots can interact with websites and download documents much faster than humans do. Moreover, bots are generally cheap to deploy, which allows running multiple bot instances per machine in parallel. This can become problematic when careless or opposing bot operators overburden a server, making web services inaccessible. Although, even performance losses without shutting down a web server is problematic as soon as other users are affected.

Cost. Visiting a website takes up network traffic and computation power at the host. This results in costs for the site provider and potential third parties on the requested page. Thus, the more bots visit a website, the higher the cost for its site provider and third parties.

Monetisation. Online advertisement is a crucial revenue stream for companies today.[35] In contrast to traditional broadcasting media, companies can track users online

[35] `https://www.statista.com/statistics/266206/googles-annual-global-revenue/`

more efficiently and may use this to deliver targeted ads. Bots pose a problem to both user tracking and advertisers. A bot visit may trigger servers to deliver ads, which are worthless when shown to a machine. Moreover, defrauding advertisers with bots can be lucrative for miscreants [Whi16]. Advertisers and analytics firms that do not invest in countermeasures risk significant financial loss and pollution of their databases.

Privacy. Web bots often target public and freely available data. Nevertheless, the use of this data can jeopardise an individual's privacy. A recent example is the case of Clearview AI, a company that collects images from the Web to offer facial recognition software to law enforcement and the private sector. Although those images are allegedly publicly available, several courts have found that Clearview AI's practice violates privacy laws across the globe. The U.K. [Inf22], Australia [Com21], Canada [Off21], and multiple countries in the EU [Int21; CNI21; Hel22; GPD22] have ordered Clearview AI to stop collecting and processing data of their citizens, including the deletion of already collected data. There have also been financial fines from €250K in Sweden[36]to €20M in Greece[37]and Italy.[38]In the U.S., the American Civil Liberties Union (ACLU) reached a settlement with Clearview AI, resulting in a nationwide restriction for Clearview AI from doing business with the private sector [ACL22].

Copyright. Collecting data from websites (scraping) is the common use case for bots. Not surprisingly, some bot-operating businesses faced allegations of copyright violations. However, rulings of copyright infringe depend on the jurisdiction and specific application. Determining whether certain web scraping activities violate copyright can be complex and has led to varying outcomes in the past. An applicable law to protect copyrights in the U.S. is the Digital Millennium Copyright Act (DMCA), which has been used against scraping companies in multiple law suites [Uni06; Uni13; Uni19]. In the EU, Directive 96/9/EC [Eur96] protects rights holders of original databases against copying and has also been the subject of scraping-related lawsuits [Cou13; Cou15; Cou21]. Recently, the EU has started to transform its laws regarding copyrights. The adoption of Directive 2019/790 [Eur19] allows text and data mining for scientific research purposes on such protected databases.

Legislation. Copyright violations are not the only accusation that web bot operator face. In the U.S., local laws like trespassing and federal laws like the Computer Fraud and Abuse Act (CFAA) have accompanied lawsuits against bot operations. Especially in the matter of web bots, courts have construed the CFAA differently within the last two decades [Sel18]. In a more recent lawsuit, the Ninth Circuit made an indicatory sentence in favour of bot operators [Uni19]. In this case, the defendant hiQ scraped public profiles from the social network site LinkedIn. LinkedIn had attempted to stop hiQ via the CFAA with a cause-and-desist letter. Still, the court granted injunctive relief to hiQ as it saw its business threatened. It further ordered LinkedIn to shut

[36]https://edpb.europa.eu/news/national-news/2021/swedish-dpa-police-unlawfully-used-facial-recognition-app_en

[37]https://edpb.europa.eu/news/national-news/2022/hellenic-dpa-fines-clearview-ai-20-million-euros_en

[38]https://edpb.europa.eu/news/national-news/2022/facial-recognition-italian-sa-fines-clearview-ai-eur-20-million_en

down installed countermeasures that hindered hiQ from harvesting public profiles from LinkedIn. The court also ruled that the CFAA does not apply and that public interest favours hiQ. However, it remains to be seen whether upcoming lawsuits or new laws will change this perspective further. For the time being, some web bot operations will continue to exist in a grey zone until challenged in court.

In contrast, the European Union is currently changing its legislation due to a switch in strategy that aims to renew regularisation concerning the digital world. Within this process, various laws will or have recently emerged: the Directive of security of network and information systems,[39] the EU AI Act,[40] the Digital Services Act and the Digital Markets Act,[41] the European Data Governance Act[42] and Data Act,[43] the Regulation on electronic identification and trust services,[44] and the Regulation on ePrivacy.[45]

The future will show whether these changes can significantly affect web bot operations.

2.3.2 Mitigating damage

Since web bots potentially cause damage to web users and site providers, the question arises to what extent automated measurements can avoid such damage. To answer this question, we explore best practices for mitigating damage and discuss them in the context of web measurements.

Indeed, there has been an early recognition of the potential damage that bots can cause. In 1993, Koster and others [Kos93] published a list of suggestions for how to be accountable and deploy bots cautiously under the title "*Guidelines for Robot Writers*". However, some website providers may prefer to define more permissive or restrictive rules depending on the bot or context. Hence, it is beneficial for both bots and sites when website providers can express rules individually. The Robots Exclusion Protocol (REP) [Kos94] describes such an approach and is currently in the process of standardisation [KIZ+22]. Due to the REP, a website can express what paths a bot must avoid by simply providing a robots.txt file[46] for download.

The REP relies on compliance, i.e., it requires the goodwill of bot operators. The same applies to ethical guidelines, however, they may cover aspects that REP does not, such as accountability. Many such guidelines proposed by organisations and practitioners exist, but there is no consent among them. The literature review in Table 2.2 lists a summary of various guidelines concerning the ethical deployment of web bots. While rules in the analysed guidelines partially overlap, they also show contradictions. Especially when it comes to hiding bots, some provoke ways to conceal a bot or a bot operator's identity (cf., source 10 and 11 in Table 2.2). At the same time, others consider hiding as unethical in general (see source 6 in Table 2.2).

[39] `https://digital-strategy.ec.europa.eu/en/policies/nis-directive`
[40] `https://artificialintelligenceact.eu/`
[41] `https://digital-strategy.ec.europa.eu/en/policies/digital-services-act-package`
[42] `https://digital-strategy.ec.europa.eu/en/policies/data-governance-act`
[43] `https://digital-strategy.ec.europa.eu/en/library/data-act-proposal-regulation-harmonised-rules-fair-access-and-use-data`
[44] `https://digital-strategy.ec.europa.eu/en/policies/eidas-regulation`
[45] `https://digital-strategy.ec.europa.eu/en/policies/eprivacy-regulation`
[46] By convention this file is located under `protocol:{domain_name.tld}/robots.txt`

Table 2.2: Literature review of guidelines for ethical crawling

	Guideline	By	Description
complying	Share results	1	Make results accessible.
	Limit selection	1,3,4,7,11	Narrow data collection to content that is needed.
	Stay alert	1,7,8	Bots may break or produce insufficient results.
	Account for errors	1,2,5	React on HTTP status codes, such as 403 (Forbidden) or 429 (Too many requests).
	Reduce load	1,2,5	Test locally, avoid re-visits, and use caching.
	Protect data	4	Deploy protection measure for statistics, personally identifiable information, etc.
	Deploy responsible	1,2,3,4,5,6, 7,8,9,10	Throttle request rate to not influence others and avoid deployment during peak hours.
	Evaluate necessity	1,3,4,6,7	Visiting a targeted is not always needed, e.g., data from the Internet archive or Google cache may be sufficient.
conflicting	Publish raw data	1	Make data public to avoid re-visits from others.
	Be available	1,4,7	Site owners may seek contact to a bot operator.
	Respect site rules	2,3,4,5,6, 8,9,10,11	Follow directions from a site's robots.txt, terms of use and avoid *nofollow* links.
	Be identifiable	1,3,4,5,6, 7,9,10	In the HTTP header, provide a unique value for the "User-Agent" field and a contact email address in the "from" field.
	Ask for permission	1,3,4,6,9	Contact the site owner and make your intentions clear. Deploy bots after approval. Provide information about automated activities on a website.
	Use APIs	3,9,10	An API allows the site owner to enforce rules, while bots act with permission.
	Pay back	3,7	Give credit to data source and link back to it.

1. Koster [Kos93]
2. https://dev.to/miguelmj/how-to-make-an-ethical-crawler-in-python-4o1g
3. https://www.empiricaldata.org/dataladyblog/a-guide-to-ethical-web-scraping
4. https://ec.europa.eu/eurostat/cros/content/WPC_ESSnet_Webscraping_policy_draft_en
5. https://www.cis.uni-muenchen.de/~yeong/Kurse/ss09/WebDataMining/kap8_rev.pdf and
6. https://medium.com/codex/guide-to-ethical-web-crawling-8b573cdfafdc
7. https://towardsdatascience.com/ethics-in-web-scraping-b96b18136f01
8. https://www.smashingmagazine.com/2021/03/ethical-scraping-dynamic-websites-nodejs-puppeteer/
9. https://soshace.com/responsible-web-scraping-gathering-data-ethically-and-legally/
10. https://finddatalab.com/ethicalscraping
11. https://dev.to/digitallyrajat/the-ultimate-guide-to-legal-and-ethical-web-scraping-in-2022-4c11

Interestingly, around half of these guidelines may be impractical for researchers due to common conditions of web studies. The section marked as *conflicting* in Table 2.2 highlights rules that commonly conflict with the requirements of web measurements. A delicate matter can be any requirement that makes a bot identifiable, as it can undermine the reliability of a study. Similarly, asking a site for permission could change the behaviour of the observed site, which can invalidate measurement results. Another matter is the usage of an API. These may offer an insufficient resolution (often enforced through request limits) or can lead to deviating results from a web user perspective. We further see that some guidelines are challenging to follow when scale is required, such as respecting the terms of use and asking for permission. In addition, researchers may not share collected raw data to protect study subjects or to avoid copyright violations. Finally, a study may have individual requirements that require further consideration of violating some of these rules, e.g., the bot operator must decide whether a site that restricts any access by bots within its robots.txt file is in the scope of its research. In general, guidelines should be carefully revisited under the specific requirements of a study. To put this discussion into the context of this

work, we can infer some generic rules, which we followed during the conduction of the research in this thesis.

Assessing risks and selecting preventive measures. It is crucial to determine the risk of a project to mitigate harm. The Open University offers two types to discuss ethical concerns in research projects: the formal Commissie Ethische Toetsing Onderzoek (cETO) and, independent from it, the Ethical Advisory Board (EAB), which is specific for the computer science department at the Open University. The latter provides advice with a fast turnaround time. In general, we sought advice from the EAB to discuss our research proposal and contacted the cETO if recommended by the EAB.

Responsible deployment. In general, we follow the rules listed under *complying* in Table 2.2. These reduce adverse effects related to cost, denial-of-service and monetisation. To the best of our knowledge, deploying bots, as done in this thesis, does not violate EU laws and follows the current practice in research at the time of writing.

Handling of sensitive research. Despite bot deployment, we see two areas of sensitive research that could require precautionary measures: involvement of human subjects and vulnerability research. The involvement of human subjects and the collection of personally identifiable information (PII) in this thesis is too little (see Section 8.1 for more details) as it would affect individuals. Nevertheless, our research may reveal vulnerabilities, wherefore we take precautionary matters. First, we do not perform any attacks on systems used in production. Instead, we collect data and conduct tests in a lab environment. Second, we report all found vulnerabilities under the guidelines for responsible disclosure as stated by the Open Web Application Security Project (OWASP) foundation [Tea21].

Chapter 3

Related Work

A related work section may serve two purposes: first, giving an overview of previous and current advances that relate to the research conducted in the present work, and second, contextualising the own achievements with the state-of-the-art. This chapter covers the former, while the latter will be explained in individual chapters. In the remainder of this chapter, we discuss research that has studied the automation of logging in on websites and platform-depended pricing in online markets (Section 3.1). Then we report on studies that have investigated bot detection techniques, their proliferation, and their impact on bots and measurements (Section 3.2).

3.1 Increasing measurement coverage

The first part of this thesis covers two areas of data acquisition for web measurements. First, we address automatic logging in to improve the scale of previous measurement studies. Logging in is a requirement to assess session security thoroughly. After a successful login, it may be possible to access the tokens used to identify an authenticated session and unlock website areas that should only be available to authenticated users. Studies that do not log in may miss such important details. Second, websites show dynamic behaviour and underlie frequent changes. They collect data about their visitors and provide a tailored response. Measuring such differences between individuals or platforms is a vivid research field. Thus, this part reports on research that measures such differences, focusing on online markets, platform-specific differences, or both.

3.1.1 Automating logging in

Automating logging in has been an attractive endeavour for researchers, but the capabilities of such systems have been limited for a long time. For example, multiple studies proposed solutions to secure session cookies [dRND+12; NMY+11; BCF+14; TDK11]. However, lacking the ability to log in automatically, none of these studies could test their solutions on authentication cookies. They all evaluated their solutions against cookies set prior to logging in. Tang et al. [TDK11] explicitly shy away from automated logging in and even see it as infeasible. Where previous studies needed to log in, they typically relied on manual intervention. While this approach enables studies to log in, manual intervention scales poorly and is an obstacle to repeatability.

Mundada et al. [MFK16] approached automation after logging in. They use manual logins to automatically analyse the security of the login process of 149 sites with a browser extension. They found several security risks in well-known sites such as Yahoo. However, their approach is only repeatable with their volunteer corps. Various steps towards more automated approaches to logging in have been made. Both the study by Van Acker et al. [vAHS17] and the work of Ghasemisharif et al. [GRC+18] needed to identify the login area. Van Acker et al. studied the security of the login area, while Ghasemisharif et al. counted the prevalence of single sign-on (SSO) providers. Both studies automated the identification of login areas using similar methods. Van Acker et al. evaluated the Alexa Top 100K and found 32K login pages vulnerable to man-in-the-middle attacks. Ghasemisharif et al. evaluate the Alexa Top 1M and found 58K websites offering SSO login.

To the best of our knowledge, only three studies before our development of Shepherd achieved some success in automatically logging in on websites. Calzavara et al. [CTB+14] used a crawler that submits credentials by taking an URL and a pair of username and password. The crawler then searches for login pages and assesses if the login was successful based on the presence of the username or the absence of login forms. This gave them access to 70 websites and the largest data set of authentication cookies at that time. They expanded this data set to 215 websites in an extended version of their previous work [CTC+15]. The two other studies used Facebook as an SSO provider to log in on websites. Robinson and Bonneau [RB14] manually performed the step of finding login pages. For their study, they collected sites that offer Facebook Connect and automatically logged in on them by using Facebook credentials. They focused on what permissions a visited site obtains to the user's Facebook profile. As such, they did not check whether the login was successful, nor did they evaluate aspects of the visited site. In contrast, Zhou and Evans [ZE14] designed an approach to automatically log into websites with Facebook and scan for SSO-related implementation flaws. Their scanner automates the search for Facebook login buttons, the submission of credentials, the eventual filling of registration forms, and the evaluation of whether a login was successful on English-speaking sites. On the U.S. Top 20K, they found 1,660 sites providing Facebook login, which they investigated. For the Top 10K, the authors report an 80% success rate of their method.

After Shepherd, Drakonakis et al. [DIP20] leveraged automatic account creation via SSO or domain-specific accounts. Automatic account creation allows targeting any site but requires significantly more scanning effort. On 1.6M scanned websites, they ended up with 25K sites where they reached post-login areas.

3.1.2 Mobile Web ≠ regular Web?

A most pertinent question is whether or not actual devices are needed for data acquisition. Several studies investigated differences between mobile and desktop sites, typically based on faking/emulation. Das et al. [DAB+18] investigated access to sensors on mobile devices via JavaScript. They modified OpenWPM [EN16], a web measurement tool based on the Firefox desktop browser, to resemble a mobile browser. Their findings show that most third-party scripts in the context of advertising or bot detection use mobile sensors. Goethem et al. [vGPJ19] took a similar approach when measuring differences in the deployment of security measures between desktop sites

and their mobile variants. To gain access to mobile sites, Goethem et al. modified a headless Chrome browser to fake the characteristics of a mobile device. Their study showed little difference between both variants in the deployed security measures. They do not report whether they validated the accuracy of their modified Chrome browser compared to sites delivered to real devices.

In contrast to these two studies, Yang and Yue [YY20] used genuine mobile browsers and modified OpenWPM to run on a mobile Firefox on real smartphones. They found that measures to disguise desktop browsers as mobile browsers can be ineffective in triggering the delivery of mobile websites. Their results also show a significant difference in the number of trackers between mobile and desktop sites.

3.1.3 Price differentiation

Multiple studies have examined whether web analytics is used to determine prices in online shops. In 2012, Mikians et al. [MGE+12] found that the user's geolocation, origin, and trained personas can influence prices in some e-commerce markets. Price studies typically require a dedicated scraper per vendor, limiting their coverage. To achieve vendor-site agnosticism, Mikians et al. developed $heriff [MGE+13], a browser extension to crowd-source price information. Later, Iordanou et al. [ISS+17] expanded this work to enable peer-to-peer price comparisons. Their method identified that 76 out of 1,994 services conduct location-based price discrimination.

Hupperich et al. [HTW+18] investigated what features of a device's fingerprint are most influential for prices. They use a desktop browser and modify the user-Agent value to emulate browsers. They found that the browser properties navigator.userAgent and navigator.vendor affect prices the most.

Hannak et al. [HSL+14] get real users involved via Amazon's Mechanical Turk. In addition, they investigated mobile browser platforms by deploying headless browsers with a manipulated user-agent string. Their experiments show signs of price discrimination and search steering targeting mobile browser users in two cases. Finally, Vissers et al. [VNB+14] conducted automated measurements to investigate whether price differences exist on airline websites. Their study used various user profiles and physical locations, but did not reveal structural price differences.

3.1.4 Online pricing algorithms

Previous studies have empirically investigated online pricing algorithms. Chen et al. [CMW15] studied swelling prices on Uber, which provided insights into the endurance and frequency of such swells, and used algorithms. To acquire the data for their study, they mimicked HTTP interaction by a mobile app to communicate with the backend, bypassing the need to scrape the app. In another study, Chen et al. [CMW16] attempt to reconstruct pricing algorithms by third-party sellers on Amazon. They fall back to using web scraping for the data acquisition, as the API provided by Amazon enforces restrictive rate limiting. Their findings show the usage of pricing algorithms in some cases and high price volatility. In contrast, Gibbs et al. [GGG+18] used data from analytics companies to study pricing algorithms on Airbnb. However, they found only limited use of dynamic pricing in this market.

3.2 Advancing attacks and defences

A website can deploy preventive measures against web bots or use bot detection followed by countermeasures, such as rate limiting or blocking.

Under preventive measuress, we understand methods that make it harder for bots to conduct certain activities at scale without detecting a bot. They typically rely on randomisation (e.g., mutating page elements, JavaScript, or CSS [VYG13; WZX+16]) or challenges that ought to be easy to solve by humans but not by machines, so-called CAPTCHAs[1] [vABH+03]. CAPTCHAs may prevent bots from accessing certain content or functionality; however, even humans can struggle solving them [BBF+10]. Hence, overusing CAPTCHAs results in a lousy user experience. Captcha-solving services further limit their effectiveness. These offer real-time solving either via human labour or automated routines. Counterintuitively, services relying on human work provide affordable prices, while the development of automatic captcha solvers offers only a low return on investment [MLK+10]. Contemporary CAPTCHA providers integrate bot detection routines [GVM+22] to determine the risk level per visitor. Based on the assigned risk, they can serve CAPTCHAs with varying puzzle strengths [SPK16c].

Bot detection itself can be separated into two classes: generic detection (e.g., based on site traversal [SD09; BLR+10], session data [XLC+18], mouse movement [CGK+13], etc.) and specific detection, which recognises distinctive aspects of one or more known web bot frameworks. Thus, specific bot frameworks may be detected due to their fingerprint (specific detectors) or interaction characteristics (generic detectors). The latter requires identifying distinctive behavioural traits in a bot's interaction with a site. We discuss advances in both fields further below.

3.2.1 Specific detection

Specific bot detection relies on browser fingerprinting. The field of browser fingerprinting evolved from Eckersley's study [Eck10] into using browser properties to re-identify a browser. He was able to reliably identify browsers using a only few browser properties. Since then, it has been studied extensively in the context of user tracking, as summarised in [LBB+20]. Using browser fingerprinting to identify specific client components (such as automation frameworks) has recently gained more attention. Shekyan [She15b] conducted manual investigations of PhantomJS to pin down identifiable properties. Likewise, Vastel [Vas17; Vas18; Vas19] manually investigated inconsistencies to detect Chrome in headless mode. Burzstein et al. [BMP+16] used canvas fingerprinting to distinguish between real and emulated devices. Hence, custom headless browsers imitating consumer browsers, such as PhantomJS or HTMLUnit, are prone to this type of fingerprint-based detection. In contrast, Schwarz et al. [SLG19] applied a new form of fingerprinting, JavaScript template attacks, to perform client-side vulnerability scanning. For creating the template, they traverse the object hierarchy and store the characteristics of each object. Later on, templates can be compared to determine the difference.

Various authors have suggested methods to spoof properties in JavaScript in response to fingerprinting. Spoofing techniques to defeat browser fingerprinting are performed directly in JavaScript [FZW15; TJM15] or on the browser level [NJL15;

[1] Completely Automated Public Turing test to tell Computers and Humans Apart

LBM17]. Similarly, spoofing allows camouflaging bots or emulating different device classes (e.g., mobile browsers [DAB+18]). The puppeteer stealth plugin[2]

is an open-source project to hide identifying properties in puppeteer-controlled Chrome browsers. Jueckstock et al. [JSS+21] used this plugin to test a lesser detectable Chrome browser against an easy-to-detect headless variant of Chrome. Also, Vastel et al. [VRR+20] experimented with spoofing in headless and vanilla Chrome to challenge bot detectors in the wild. They found that these scripts rarely detect vanilla Chrome. Finally, Cassel et al. [CLB+22] and Kuchhal et al. [KL21] created self-made automation components to avoid detection via known properties in automation frameworks.

3.2.2 Generic detection

There has been a vast number of studies that explored site traversal and session data for bot detection. While these approaches achieved satisfying results within their experimental boundaries, it is unclear which approach works sufficiently well in practice [DG11]. Another research direction in this area is detection based on interaction. Various approaches use interaction as a means to re-identify individuals. In particular, researchers have commonly used interaction as a complement to passwords. Therefore, these studies focus primarily on typing rhythms as a secondary authenticator [RSJ20]. This idea was also adapted to mobile phones. Giuffrida et al. [GMC+14] used touch events and enhanced this with data from smartphone sensors (e.g., accelerometer and orientation sensor). In contrast, studies have also examined automating interaction to perform credential stuffing or brute force attacks. Such works challenge the security of interaction detection systems by imitating human interaction with bots. For example, the study by Serwadda and Phoha [SP13] demonstrates such an attack by using a large-scale data set containing keystrokes from real users to simulate human typing. Similarly, Serwadda et al. [SPW+16] brought this attack to mobile phones by leveraging a robotic arm to mimic gestures.

Bots are used in other circumstances to circumvent expectations. This is particularly egregious in online games (e.g., aim bots) and online discussion fora (e.g., spam bots). The academic community has investigated these aspects at great length. For example, Barik et al. [BHR+12] used mouse button presses and release events to detect automated clients in web browser games. Gianvecchio et al. [GWX+09] used the limited variety of player interaction to identify bots in the context of an MMO. Park et al. [PPL+06] assumed that bot interaction on websites is inconsistent. Hence, they proposed bot detecting based on missing actions (e.g., the lack of mouse movement when performing a click). Chu et al. [CGK+13] used keystroke dynamics and mouse movement to identify spam bots on blogs. Detection of such bots is also a hot topic outside academia. As such bots cause significant annoyance for users and maintainers/moderators, a plethora of tools, blog posts, and other non-peer-reviewed discussions exist that attempt to identify such bots. On the other hand, such bots can offer significant benefits to their users, who actively discuss ways to introduce more realistic interaction into them.

[2] `https://www.npmjs.com/package/puppeteer-extra-plugin-stealth`

3.2.3 Measuring bot detectors in the wild

Measuring the Web for bot detection provides insights about the state-of-the-art. Besides our work (see Chapter 7 and 9), only a few studies have conducted such measurements. Das et al. [DAB+18] studied access to mobile sensor APIs in browsers and found that scripts belonging to bot detectors make use of them. Vastel et al. [VRR+20] counted bot detectors by detecting incidences of blocking. Their study shows a rate of 2.9% of sites within the Alexa Top 10K with detectors that block a detected bot. Azad et al. [ASL+20] evaluated whether third-party bot detectors prevent websites from automated attacks (account takeover, brute forcing and automatic scraping). As they needed a seed list of sites protected by detectors, they created code signatures of bot detection services and looked for these signatures via search engines. Another study by Jueckstock and Kapravelos [JK19] contains a large-scale investigation of the existence of unknown fingerprint-based bot detectors. They presented an experiment using dynamic script collection and dynamic analysis. For that, they created VisibleV8 (or just VV8), a modified version of Chrome's V8 JavaScript engine to record JavaScript calls. Their instrumentation allows recording attempts to access non-existing properties in the DOM without exposing such properties to page scripts.

3.2.4 Are web bots served the same content as humans?

Websites may return a different page to a web bot than a regular visitor would receive. This practice is commonly referred to as *cloaking*. Cloaking techniques fall into the category of dual-use technology. For example, malicious actors use cloaking for targeted attacks, such as search engine manipulation [WD05; WD06; WSV11; ITK+16] or to protect phishing websites from web security scanners [OSD+19; OZW+20; ZOC+21]. On the other hand, websites may adjust their behaviour to reduce costs or to shield themselves against automated attacks. For example, Englehardt and Narayanan [EN16] noticed that websites serve fewer ads to PhantomJS. Moreover, they found that they would get more ads if they changed the UserAgent string to that of a regular user browser (Firefox). As this raises concerns regarding the reliability of measurement tools, researchers have investigated measurement deviations between different bots and between humans and bots.

Ahmad et al. [ADZ+20] contrasted response differences between three classes of web bots (HTTP engine tools, headless browsers, and automated browsers). They found that while HTTP engine tools miss many resources, they more often pass bot detection than the other two classes. Pham et al. [PSF16] focused on deviations originating in the UserAgent string. By alternating the UserAgent string of an HTTP engine, they observed that strings referring to web bots resulted in significantly more (about 4×) HTTP error codes than strings of regular browsers. Interestingly, UserAgent strings of a relatively unknown web bot framework worked better than an empty string. They even outperformed the strings of regular browsers. Jueckstock et al. [JSS+21] studied differences for headless Chrome and IP origins, such as universities, cloud and residential IPs. Their study shows significant differences in website responses between a headless browser and a less detectable, scripted Chrome browser.

In contrast, Zeber et al. [ZBO+20] compared data from human users with OpenWPM clients. In their study, OpenWPM clients encountered three times more tracking domains and had more interaction with third-party domains than human-

controlled browsers. Cassel et al. [CLB+22] investigated the reliability of emulated browsers. To avoid bot detection, they created their own browser remote control. Interestingly, their observations show the opposite of Zeber et al.'s findings. They observed 84% less third-party traffic for a Selenium-driven vs a non-Selenium-driven Firefox browser. This contradiction shows that there is yet no consistent picture of the influence of bot detection on measurements.

Part I

Increasing Measurement Coverage

Chapter 4

Overcoming the Login Barrier

To gauge adoption of web security measures, large-scale testing of website security is needed. However, the diversity of modern websites makes a structured approach to testing a daunting task. This is especially a problem with respect to logging in: there are many subtle deviations in the flow of the login process between websites. Current efforts investigating login security typically are semi-automated, requiring manual intervention which does not scale well. Hence, comprehensive studies of post-login areas have not been possible yet.
In this chapter, we introduce Shepherd, a generic framework for logging in on websites. Given credentials, it provides a fully automated attempt at logging in. We discuss various design challenges related to automatically identifying login areas, validating correct logins, and detecting incorrect credentials. The tool collects data on successes and failures for each of these. We evaluate Shepherd's capabilities to login on thousands of sites, using unreliable, legitimately crowd-sourced credentials for a random selection from the Alexa Top websites list. Notwithstanding parked domains, invalid credentials, etc., Shepherd was able to automatically log in on 7,113 sites from this set, an order of magnitude beyond previous efforts at automating login.

This chapter is based on the following publication:

Shepherd: a Generic Approach to Automating Website Login. Hugo Jonker, Stefan Karsch, Benjamin Krumnow, and Marc Sleegers. *In Proc. 2nd NDSS Workshop on Measurements, Attacks, and Defenses for the Web (MADWEB'20)*, DOI: 10.14722/madweb.2020.23008, 2020, [MADWeb20].

4.1 Introduction

Security of online services must be regularly tested. This is not only needed to improve security of specific services, but also to gauge the state of adoption of security measures. For websites, an interesting paradox presents itself: a major security aspect is the login process, with further security aspects of interest accessible only to logged-in users. However, the login process may vary from website to website. Thus logging in automatically across a wide variety of sites is a daunting challenge – one we address in this chapter.

Websites offer users the option to login, typically for one of two reasons: to access protected resources (such as a personal mailbox), or to participate in the website's community under a specific identity. In either case, security of the authentication process is of fundamental importance. Websites should ensure that an unauthorised attacker cannot steal or overtake the login (session hijacking). However, websites are often vulnerable to simple session hijacking attacks.

For example, in 2010, the Firesheep browser plugin for Firefox [But10] trivialised one class of (already known) session hijacking attacks. While this one attack can easily be prevented, authentication cookies can still be stolen or leaked in a number of other ways. Nowadays, several simple mitigation measures exist which can be used to prevent a whole range of simple attacks against authentication. These include cookie flags that restrict when a browser sends a cookie, HTTP headers that enforce secure communications for all subsequent visits, etc. Sites that lack these measures are vulnerable to simple session hijacking, while sites that do have them will offer a base line of security. Manually assessing the security of a specific site is straightforward. Indeed, due to manual verification, we know that the sites affected by Firesheep shored up their defences. However, applying this process to all websites does not scale and is thus typically not performed. Case in point: we do not even know how many other sites are still open to the Firesheep attack – or simple variations thereof.

A similar open question concerns the uptake of modern security measures. For example, we do not know how many sites lack basic security measures (proper cookie flags and HTTP headers) for logged in users. Other open questions concern adoption of security and privacy-enhancing measures beyond those affecting session security, for which logging in is a prerequisite. To study such questions requires two ingredients:

1. a set of valid credentials,
2. successfully submitting those credentials.

While several efforts have investigated specific security aspects on a handful of sites, to date, most studies that evaluated the security of the authentication process relied on a combination of automation and manual labour or (like Firesheep) tailored their measurement to specific websites (see Section 3.1.1). Typically, the manual aspect focused on actually logging in. This presents a barrier to scaling up these investigations and addressing the aforementioned questions. To the best of our knowledge, the largest manual study to date that successfully reaches post-login stages used manual logins to evaluate 149 sites. Initial attempts at automating the login process relied on single sign-on (SSO) credentials (such as Facebook login) and reported success on 912 websites.

Contributions. The goal of our work is to study the feasibility of large-scale post-login studies without tailoring the automation to a specific login flow. To that end, we make the following contributions.

- We present Shepherd, a framework for automatically logging in on websites and executing a scan.
- We identify challenges in identifying login areas, on validating correct logins, detecting invalid credentials, and provide approaches to handling each of these.
- We perform a scan to illustrate Shepherd's potential.
  Using credentials gathered from a legitimate, crowd-sourced effort, we successfully login on 7,113 websites. The case study shows Shepherd is able to study a number of sites an order of magnitude beyond any previous study.

Outline. In the following, we discuss Shepherd's routines to automatically log in on websites (Section 4.2) and shine a light on some implementation details (Section 4.3). Then, we review approaches for creating a data set with credentials and use one to evaluate Shepherd's performance (Section 4.4). Next, we validate our measurement (Section 4.5) and compare Shepherd's performance with previous studies (Section 4.6). Before concluding this chapter (Section 4.9), we report on possible use cases for Shepherd (Section 4.7) and present extensions to broaden Shepherd's scope (Section 4.8).

Ethical considerations. For our study, we aim to achieve an unprecedented scale in entering restricted areas of websites. As this led to concerns, we sought and received approval from our EAB. Nevertheless, we wish to highlight the various ethical concerns.

The primary concern was acquiring a large set of credentials from a legitimate source. Fortunately, the BugMeNot database – the data source we use in our study – is exactly this: a large set of login credentials with strict (and enforced) policies to ban a site from inclusion upon request of the site owner.

Secondly, the experiments must not exceed their mandate and break things. Hence, we design Shepherd to interact with websites in the same fashion as humans do: to trigger elements human clicks are simulated, timeouts are used between each action, sites are not crawled in parallel, only visible elements are considered for interacting with a site. Therefore, the risk of overburden or accidentally confusing the websites logic is reduced. We worked on this by testing Shepherd on a small number of domains and resolving any issues. The results are not 100% perfect, but the fraction of mistakenly pressed buttons we detected is very small.

Availability. The tools created can easily be misused, e.g., to apply credential stuffing or password guessing attacks. Therefore, we cannot and will not publicly release Shepherd. On the other hand, we welcome interest from fellow researchers. Thus, we make Shepherd available for follow-up studies by other bona fide researchers upon request. We list criteria that we use to evaluate requests on our project site.[1]

[1] `https://bkrumnow.github.io/shepherd/`

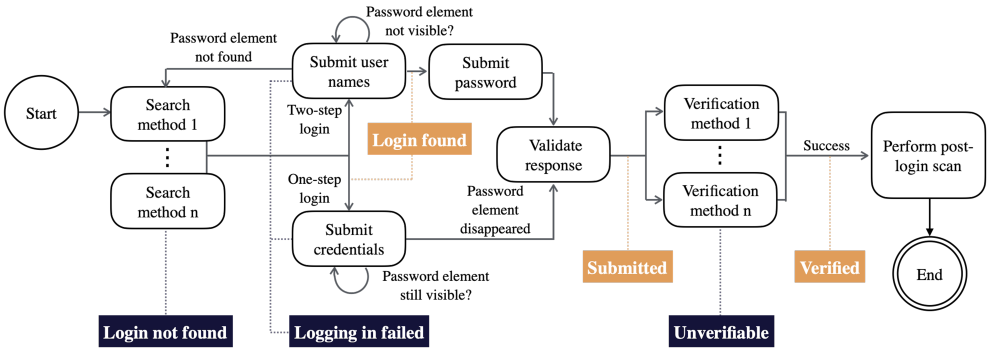

Figure 4.1: Steps of the login process after connecting to a target site

4.2 Automating logging in

Logging in is basically a sequential process, consisting of a number of steps (see Figure 4.1). To log in on a website automatically, Shepherd conducts four steps:

1. identify the login starting point,
2. submit credentials,
3. check response to login attempt,
4. verify whether login was successful.

In addition to these steps, Shepherd also detects and keeps track of certain errors. This is because Shepherd uses a generic approach, which is not tailored to any specific login process. As such, it may make mistakes (login field not found) or encounter errors from external sources (site unreachable, CAPTCHA, invalid credentials). In effect, the process acts like a funnel, with each step acting as an imperfect filter. To gauge the accuracy of the filters themselves, Shepherd includes routines to detect a variety of errors.

In the rest of this section, we discuss steps and error detection in more detail.

4.2.1 Identifying the login starting point

First, the login starting point of the target website must be found. Zhou and Evans [ZE14] approached this by relying on click events on release to trigger SSO login dialogues. In contrast, domain-specific logins may also be found by visiting URLs. From previous studies [CTC+15; vAHS17; GRC+18] five search strategies emerge: scanning the landing page, visiting URLs filter by login keywords, querying search engines, try standard URLs, and scanning clickable DOM elements. We found that combining multiple search engines can lead to a better coverage, thus we expanded the approach by Van Acker et al. [vAHS17] on this point.

Unfortunately, none of these studies provide insight in the efficiency nor reasoning about the order of these methods. To test the success of each method for finding login elements, we applied each to a random sample of 5,000 sites from the Alexa Top 1M

Table 4.1: Performance overview of methods to locate a login page of a website

	method	**success** (n=5,000)	**average time** (in sec)
1.	landing page	225	7.17
2.	URLs with login terms (depth 1)	1,119	10.86
3.	clickable elements	1,149	18.84
4.	standard URLs	1,366	18.79
5.	search engines	1,948	50.08
	- startpage.com	1,378	32.66
	- bing.com	342	9.17
	- ask.com	1,216	18.04
	All combined	**2,759**	**36.43**

(see Table 4.1). We found that landing pages rarely contain login elements. However, all other methods rely on the actual domain of the site (after any redirects). Therefore, the method of scanning the landing page should be executed first. Furthermore, some methods are more successful in finding login pages than others. Interestingly, the results of the various methods are sufficiently disjoint that combining them leads to the highest success rate.

Based on the evaluation, we arrived at the following order to search for login elements:

1. Landing page,
2. URLs with login-based terms found on the landing page,
3. Clickable elements with login-based terms,
4. Standard URLs[2],
5. Search engines,
6. URLs with login-based terms found on pages from step 2.

The order of these methods is important. Shepherd looks for login elements on the landing page first, since that page needed to be loaded anyway. Only when method 2 and 3 fail, Shepherd uses more generic methods. We gave standard URLs a lower priority, as these can lead to admin login pages. Shepherd only uses search engines if prior methods fail to reduce the risk of blocking and reliance on external parties. Finally, if none of these methods work, Shepherd scans each of the pages found in step 2 for URLs with login-related terms and visits these. For search terms, Shepherd contains a dictionary with multiple translations for keywords from native speakers and Google translations.

Once a method claims success, Shepherd stops searching for the login. If none of the search method worked, Shepherd finishes the scanning process and marks that a login page could not be found.

When Shepherd encounters a visible input element of type password which is not part of a registration form, it assigns the status *login found*. A form is considered

[2]Specifically: http(s)://base_url/login, http(s)://base_url/account and http(s)://base_url/signin.

a registration form if it contains more than 3 visible input elements (including the found password field).

4.2.2 Submitting credentials to login

Once the login element has been found, the credentials must be submitted. There are two common types of logins:

- one-step: where username and password can be supplied simultaneously; and
- two-step: first request the username, and only after the username has been submitted is the password entry field shown.

By combining logging in with scanning for login areas, we are able to cover both types. To the best of our knowledge, submitting credentials for two-step login has not been explored in previous work.

If fully submitting the credentials (in either one-step or two-step fashion) causes the password input element to disappear, the website status is set to *submitted*. This status indicates that the site responded to the input, but does not claim that login was successful. Websites can also stop showing password fields in other cases, such as when the user is blocked or an error results in a 404 page. To separate such cases from actual logins, the submission process is followed by a verification process.

Shepherd can use multiple credentials per domain (there is also limited support for Facebook SSO credentials, as described in Section 4.8.1). If it is not successful with the first set of credentials (i.e., the password field remains), Shepherd will try logging in anew with the next set of credentials. In case all available credentials for a specific domain fail, the website is assigned the status *logging in failed, after trying all credentials*.

Finally, in some login forms, the username is an email address. If the input element is of type "email", Shepherd avoids submitting strings that are not valid email addresses. For cases where Shepherd can ensure that an email address is required to log in, given credentials without an email address will be ignored.

4.2.3 Checking the response to a login attempt

Shepherd checks the website response after submitting credentials in order to perform some error detection. For example, many websites signal a failed login attempt with a message like "username or password invalid." Shepherd detects such messages and marks the credentials as invalid. More specifically, Shepherd assesses a website's language and searches for visible strings containing sets of keywords, such as "invalid" and "username" or "password". Only if these terms appear combined in a single string, Shepherd marks the credentials as invalid. In practical tests, failure messages occasionally appeared in English on non-English sites. Thus, Shepherd scans for keywords concerning invalid credentials in English and the site's used language. Besides invalid credentials, Shepherd also detects CAPTCHAs and blocking messages to recognise countermeasures against automated visitors. To do so, Shepherd scans visible elements for blocking or CAPTCHA related keywords in multiple languages, and checks the HTML source for code fragments pertaining to CAPTCHAs. The code fragments

Shepherd detects are derived from the HTML code fragments used to invoke one of five CAPTCHA libraries (including ReCaptcha).

4.2.4 Verifying login status

After detecting that the website reacted to the submission of credentials, Shepherd evaluates whether logging in was successful. For this, we use previous approaches [CTC+15; ZE14; MFK16] with a few differences. In previous studies, trusted credentials or manual logins were used. In this work, we assume credentials to be unreliable. Moreover, we also do not assume that the occurrence of a string that matches the username on a page is sufficient to verify login – strings occurring in usernames may also occur on the page due to other reasons.

To verify a login, Shepherd runs a verification method twice: once on the potentially logged-in site, and once without cookies. Login is only successfully verified if the first check succeeds and the second fails.

Shepherd has three verification methods used for this:

1. detect a logout button or user identifier on the page received following submission of credentials,
2. detect a logout button or user identifier on the landing page,
3. attempt to re-open the login area.[3]

A login is only claimed to be verified if at least one of these verification methods is successful when visiting the site with cookies, and fails when visiting the site without cookies.

4.2.5 Perform post-login scan

Following a successful verification of logged-in status, Shepherd will execute any scans. The process to hook in custom routines is described in Section 4.3.4.

4.3 Implementation

Shepherd uses Chrome as a designated browser, and runs on Linux and on macOS systems. The instrumentation is achieved through usage of Selenium.

4.3.1 Base HTTP platform

There are several possible platforms on which to build Shepherd. Not all are suitable. HTTP engine interfaces lack engines to interpret JavaScript and construct DOMs, which is necessary for websites with dynamic content. More advanced tools, such as custom browsers (e.g., Nightmare) or headless browsers, are an improvement on this but nevertheless still lack some of the functionality of full web browsers (e.g., plugins). This poses two problems: firstly, they do not necessarily provide a faithful rendition of what a regular user would experience (cf., [EN16]); secondly, such deviations can

[3] In well-designed sites, this should not be possible for logged-in users.

affect logging in (cf., Section 7.6). Thus, we require an automated way to use a regular web browser. Shepherd uses a standard automation tool for web browsers, Selenium (see Section 2.2.2), to programmatically access browser instances.

4.3.2 Speeding up page analysis

Logins are typically slow and can easily take several seconds. When attempting to login on unknown sites, using a form which may or may not be the login form, with credentials that may or may not be valid, several passes have to be taken. When executing a study over many sites with all these factors in mind, performance becomes an important factor.

With respect to optimisation, we found that Selenium's built-in functions are slow compared to executing the same functions in JavaScript. For example, we measured that accessing the plain HTML content of elements takes 14 msec using Selenium functions. When operating with a large number of elements, this becomes a time-consuming operation. Another example is Selenium's function to query multiple elements *find_elements()*, which takes 1 second per query. Combining that with additional filtering based on an element's attributes or content results in a large overhead. Therefore, we switched from using Selenium's functions to using in-browser JavaScript.

To this end, Shepherd provides two JavaScript functions, *href_scanner()* and *element_scanner()*. These functions allow efficient selection of anchor and other elements. The former function searches amongst anchor elements with `href` attributes, while the latter can select any element through a custom CSS3 selector. Both scripts take a regular expression to filter selected elements based on their HTML content (e.g., login-related keywords).

Using these in-browser functions instead of Selenium functions provided a noticeable speedup. For one site, switching to JavaScript functions improved time for accessing and filtering elements from 16.8 seconds to 50 msec.

4.3.3 Runtime performance

With the above measures in place, Shepherd needs about 75 seconds to scan a site. Thus, Shepherd can scan and login to about 1,500 sites per browser instance per day. In our experiments, we found that a regular end-user machine can run five browser instances, so Shepherd can scan about 7,500 sites per day per computer.

4.3.4 Post-login scanning

Following login, payload scans (implemented as Python modules) are executed. Shepherd provides an interface to interact with the browser and detect effects of interactions. This interface is a wrapper of Selenium commands, but streamlines error handling and ensures performance-optimised commands are used by the scanning module. In addition, Shepherd offers functionality to determine which cookies are authentication cookies based on algorithms used in earlier work [MFK16; CTB+14]. Furthermore, additional modules can be hooked into Shepherd. This allows for sequential execution of several scanning modules. Scan results are determined on the fly and stored in CSV files for a posteriori analysis.

4.4 Evaluation: logging in on websites in the wild

Next, we evaluate Shepherd's ability to log in by means of a large-scale experiment.

4.4.1 Access credentials

The mandatory requirement for logging in across many websites is valid credentials. However, for legal and ethical reasons, leaked credentials cannot be used in our research. That leaves the following approaches to be considered:

1. using single sign-on (SSO)
2. automating registration
3. crowd-sourcing credentials

Note that none of these approaches will work flawlessly on all sites; each of these therefore introduces a bias in the set of sites covered by it. Some of this bias will be inherent to automated logins: credentials for, e.g., banking sites are not legitimately available at scale. Other bias will be specific to each approach.

Using SSO to log in is supported on 6.3% of the Alexa Top 1 Million [GRC+18]. SSO offers a clear advantage for large-scale studies, i.e., only a limited number of credentials are needed to log in on many different sites. Unfortunately, using SSO is also challenging: it may necessitate additional actions, such as account registration despite SSO access and authorization granting, e.g., in the case of OAuth 2.0. This makes using SSO rather hard. For example, Zhou and Evans had limited success [ZE14]: 912 logins out of 20K sites (4.6%). In addition to those challenges, using SSO imposes its own bias on the set of sites: first, sites may insist upon their own account registration system and not offer any other login (e.g., webshops, banks). Other sites may not offer SSO for privacy reasons (e.g., adult entertainment). All such sites are excluded from an SSO-only approach. Moreover, there is no single, world-wide most popular SSO provider. Different regions prefer different SSO providers. Using common western SSO providers would bias the study towards their sphere of influence; minimising such bias necessitates a world-wide view on all SSO providers and their sphere of influence.

Automating account registration may address such concerns. A significant benefit of this approach is its general applicability, as it does not require SSO availability. In a later study [DIP20], this approach was used to login on 23,176 sites (out of 1.6M sites, 1.6%). A major downside to automatic registration is that the registration process is a critical security feature of websites frequently targeted for automated attacks. As such, it is typically protected against automated visitors (e.g., by means of a CAPTCHA). Automating circumvention of techniques deliberately employed to prevent automated registration poses serious ethical concerns. Moreover, even if the ethical issues are ignored (we stress: they should not), automated registration still introduces a bias: it will only succeed on sites with insufficient defences against it, thus likely skewing towards websites with weak security.

Using **crowd-sourced credentials** from public databases solves the ethical issues related to automated account registration. Nevertheless, this also leads to a bias. The bias inherent in legitimate crowd-sourced credentials is due to the type of accounts

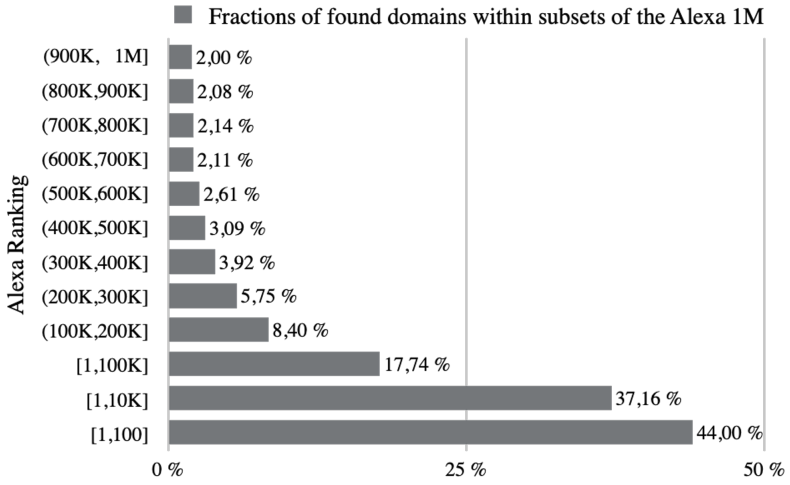

Figure 4.2: Relative frequency of domains with credentials of our testing database within the Alexa Top 1 Million

that users are willing or allowed to share. For example, sites where registration is simple and accounts are not associated with (personal) value will be prevalent, while other accounts (banks, social media, online stores), will be underrepresented or even absent due to the rules governing the crowd-sourcing effort.

To sum up, while automating registration managed (so far) to log in on the largest absolute number of sites, its success rate is an abysmal 1.6% [DIP20]. Moreover, automated registration might violate existing terms of services, while still skewing the set of sites under consideration towards weak security. Using SSO is a more viable option, but requires a complex automation infrastructure to perform an open-ended scan with a low success rate (best success rate: 4.6%). In contrast, the use of crowd-sourced credentials minimises the scanning effort, as only sites with suspected logins need to be scanned. We acknowledge this approach might still suffer from a bias coming from the availability of credentials, which however is still not entirely solved by competitor approaches. Thus, we report on several experiments designed to mitigate the impact of such bias in the following.

4.4.2 Acquiring credentials

We created a specific crawler to extract credentials from BugMeNot. The crawler uses a list of domains and for each domain, extracts the credentials. Moreover, for each set of credentials, it also stores meta-information supplied by BugMeNot (success rate and number of votes). We seeded our crawler with the Alexa Top 1 Million sites of October 2018. This resulted in the extraction of 129,252 accounts for 49,846 unique domains.

The collected data set covers over 37% of the Alexa Top 10K domains and around 18% of the Top 100K, respectively (see Figure 4.2). The concentration of websites decreases with the rankings and appears to converge around 2K domains per 100K websites.

Table 4.2: Failures detected by Shepherd. Failures caused by external factors are marked in bold.

	# sites	out of	
total	49,846	–	
sites not reached	**976**	49,846	(**2.0%**)
login page not found	10,439	48,870	(21.4%)
login failures:			
- **invalid credentials detected**	**23,088**	38,431	(**60.1%**)
- **CAPTCHAs**	**1,497**	38,431	(**3.9%**)
- unaccounted failure	2,783	38,431	(7.2%)

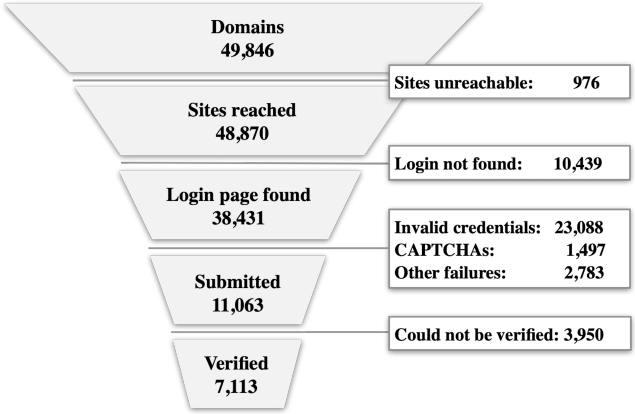

Figure 4.3: Funnel of the login process for domain-specific credentials

4.4.3 Shepherd's login performance

Using data set with credentials from BugMeNot, we can measure success rate and error causes. Starting with credentials for 49,846 sites, Shepherd was able to automatically detect that all available credentials for 23,088 sites were rejected by the site as invalid. This leaves 26,758 sites to attempt login. Shepherd could verify successful login on 7,113 sites, i.e., 26.6%. This is a lower bound: there will be external sources of errors that Shepherd failed to detect. For example, it is not certain that all websites in the set offered the option to login. Moreover, in some of the 3,950 cases where Shepherd managed to submit credentials, it may have been successful but failed to verify this. While this leaves ample room for improvement, this case study is of unprecedented scale – easily an order of magnitude beyond any previous studies.

As discussed before, logging in is a sequential process, which means that an automated approach must execute sequentially. Imperfections in each step result in a funnel-alike propagation through the login sequence, depicted in Figure 4.3. The rates shown in Figure 4.3 were automatically detected (see also Table 4.2). Of course, not all failures can be automatically attributed; for example, failing to reach the step *submitted* can be due to CAPTCHAs or invalid credentials, both of which are automatically detected by Shepherd. These error sources account for the bulk of the failures

for moving from *login page found* to *submitted*. Nevertheless, there are 2,783 websites where this transition failed, yet the built-in failure attribution did not detect a cause.

Some of the failures are due to external causes, while other causes denote potential areas for improvement of Shepherd. Failures on 25,561 sites (51.3%) were attributable to external factors: site unreachable, no valid credentials, or CAPTCHAs. The major area for improvement is identifying the login starting point, which failed on 10,439 sites (20.9%).

4.5 Validation of Shepherd

As we rely on an untrustworthy source of credentials, it is not certain that a website for which we possess credentials actually has a login facility. Also, our underlying heuristics are not 100% perfect and may occasionally fail to determine the status correctly. To determine bounds on the error rates, we manually evaluated the following five cases:

1. Failure to find login page,
2. not having reached status *submitted*,
3. detecting invalid credentials,
4. not reaching status *verified*,
5. reaching status *verified*.

We manually validate Shepherd's login procedure by creating five sets of 100 websites. In the first case, we manually visited sites, while for cases 2–5 we reviewed automatically created screenshots. In all cases, the evaluator was at liberty to skip sites containing adult content.[4] These cases give insight into performance of the heuristics and suggest which gains can still be made by improving heuristics.

4.5.1 Finding login pages

This case concerns websites where Shepherd was unable to find login areas. We evaluated 100 such sites and manually identified login areas on 44 sites. On not all of these, it was clear that the login area would provide a login for the initially visited site. One site was not manually evaluated, as it contained adult content. We did not discover login elements on the other 55 sites. Shepherd thus failed to identify login fields that were present on 44 out of 99 sites where it did not find login elements. Optimistically viewed, this can be generalised, which means that at most 9.5% more sites could be reached. For this data set, that means that at best, $10,439 \cdot 44/99 = 4,640$ more sites could be included.

[4] as such sites are more likely to contain illicit material.

4.5.2 Submission failures

This case concerns sites where Shepherd found a login area and submitted credentials, but could not detect success. We reviewed 100 screenshots of login areas of such sites. On 85 screenshots we derive that Shepherd found the correct login area. Another 8 show indications but raised uncertainties, as the login area was covered by a CAPTCHA or pop-ups in a different language. Another 5 cases were failures, where Shepherd focused elements belonging to registration elements (3) or ended up on age verification pages (2), instead of login elements (which were also present).[5] Two cases could not be evaluated due to overlaid content or the site's language.

4.5.3 Detecting invalid credentials

This case concerns Shepherd's ability to identify invalid credentials. We used the 100 screenshots from case 2. In 87 screenshots Shepherd correctly noted messages signalling invalid credentials (66) or the absence of such signals (21). In 8 cases Shepherd misclassified the responses from websites. The remaining 5 screenshots were not used, as these showed registrations, were overlaid with pop-ups or not interpretable due to the language. We conclude that given a login area, the process of credential submission and evaluation of the success of that step performs reasonably well.

4.5.4 Submitted, but unverified

This case concerns sites where Shepherd successfully submitted credentials, but failed to verify it. Note that verification is supposed to fail when login was not successful. In 76 out of 100 cases, this was the case, underscoring the need to verify whether login is indeed successful. 21 cases showed clear signs that Shepherd entered the post-login stage, while 3 cases could not be verified. In other words: in at least 21% of the examined cases, this process resulted in a false negative. In the experiment, of the sites where credentials were successfully submitted, on 3,950 sites this could not be verified. 21% of this is 829 sites. These 829 sites are sites where logins were potentially successful, but not detected by Shepherd. Additional or improved verification methods thus may lead to hundreds of sites more evaluated.

4.5.5 Verified

This case concerns 100 sites which passed verification. Of these, only one site could not be checked. Two other screenshots showed that the user account was banned. Nevertheless, Shepherd was clearly logged in on these sites. For that, we find the verification process to have high accuracy ($\geq$97%) and therefore have high confidence in all findings on sites marked *verified*.

[5] This sometimes happens when sites offer login elements outside of a form element. For such cases, other input elements can be confused with the login elements. Shepherd will use heuristics to select the most likely login related elements.

4.5.6 Performance of login-finding methods

Finally, we zoom in on the detection of the login area. In particular, we investigated the success rate of the different approaches to finding login elements. We remark once again that these methods are executed sequentially: only if methods 1–5 fail, method 6 is executed. Of the 38,431 domains where a login area was found, method 2 was most successful, finding more than 40% of login areas.

Method 1 - Landing page:	6,311
Method 2 - URLs (first level):	15,438
Method 3 - Clicking elements:	4,004
Method 4 - Try standard URLs:	3,745
Method 5 - Search Engines:	8,875
Method 6 - URLs (second level):	76

In conclusion, Shepherd managed to successfully login on 7,113 sites (submitted and verified). We found that the heuristics used in Shepherd can also be improved. Main areas for internal improvement are:

- improving identification of login elements.
  This can lead to (at most) 44% more sites reached, or 4,640 additional sites in this experiment.
- reducing false negatives for verification.
  This can lead to (at most) 21% sites where submission was successful, or up to 829 additional sites in this experiment.

However, the number one area for improvement is: valid credentials for more sites. Shepherd's detection of invalid credentials found that for 60.1% of sites where login elements were found, no valid credentials were available (23,088 sites in this study).

4.6 Login performance comparison with previous work

Various steps towards more automated approaches to logging in have been made (cf., Section 3.1.1). We contrast the performance of these earlier approaches with the performance of our evaluation of Shepherd. Table 4.3 contains login/session security related studies which either do or do not provide login automation. Van Acker et al. [vAHS17] and Ghasemisharif et al. [GRC+18] evaluated aspects on login pages. As a result, they reach a large number of sites but they do not reach post-login areas. Thanks to user participation, Mundada et al. [MFK16] manage to conduct post-login security scans. At the same time, their study reaches a far smaller number of logins. To the best of our knowledge, only three earlier approaches achieved some success in automatically logging in on websites. Robinson and Bonneau [RB14] manually selected login pages that facilitate Facebook Connect to submit credentials on these sites automatically. In contrast, Calzavara et al. [CTB+14] used a crawler that submits credentials on websites with previously created usernames and passwords. They conducted two studies with the same methods; reaching 70 sites the

first time, and 215 sites in the follow-up study [CTC+15]. The so far largest study was conducted by Zhou and Evans [ZE14]. Note that Zhou and Evans do not provide a number of reached post-login areas, wherefore we extrapolate this number from the success rate and websites with SSO login areas reported by the authors. Based on our experiment, Shepherd shows to outperform previous approaches in the number of successfully reached post-login areas.

4.7 Potential use cases

In this section, we highlight use cases for Shepherd. In general, we see two potential areas where Shepherd can boost security and privacy research. These are the investigation of post-login features across multiple websites, and the comparison of pre- vs. post-login aspects.

Measuring post-login features. By logging in, Shepherd receives authentication tokens from websites, which allows studying the security of such tokens. Shepherd possesses an implementation to extract authentication cookies, facilitating investigations into properties of these. Furthermore, Shepherd could be extended to identify sites whose login system exhibits specific behaviour. One example could be sites that store session identifiers in local storage instead of using cookies; another is looking for sites vulnerable to sub-session hijacking [CRB19] (which relies on presence of multiple authentication cookies).

Moreover, the automatic identification of authentication cookies enables construction of a ground truth for machine learning purposes (e.g., [CTB+14]). The most extensive set of authentication cookies reported in literature amounts to cookies from 215 sites (332 authentication cookies) [CTC+15]. Using the authentication cookie identification mechanism of Shepherd, we collected a set of authentication cookies for 6,335 sites – the automated mechanism failed to identify cookies on 778 out of the 7,113 sites where login was successfully verified.

Finally, Shepherd enables measuring adoption of security measures for logged-in users, such as cookie flags such as *SameSite*, *Secure*, etc.; HTTP-headers such as HSTS; CSRF tokens [CCF+19]; etc.

Table 4.3: Comparison of manual and (semi-)automated login studies

	lacks automated login			*has automated login*			
	[vAHS17]	[GRC+18]	[MFK16]	[RB14]	[CTC+15]	[ZE14]	**Shepherd**
automation:							
- finding login area	✓	✓	–	–	✓	✓	✔
- filling credentials	–	–	–	✓	✓	✓	✔
- verifying logins	–	–	✓	–	✓	✓	✔
# supported languages	10	?	?	?	?	1	**19**
# sites scanned	100K	1M	149	203	215	20K	**49K**
# successful logins	n/a	n/a	149	203	215	912†	**7.1K**

? number of supported languages unknown.
† computed, as [ZE14] does not explicitly state this number.
n/a not applicable.

Anonymous visitors vs. logged-in users. The above suggestions for measurements can also be applied *before* logging in. Hereby, Shepherd enables studying the contrast between anonymous visitors and logged-in users on a large scale. For example, logged-in users may face less trackers than anonymous users. An additional direction is to use Shepherd to create authentic user profiles by gathering cookies from several sites. User profiles have already been used in earlier research to analyse ads [CMC+15] or evade bot prevention measures [SPK16a]. Finally, Shepherd could be extended to automate logging out. This would enable a large-scale study comparing the session state while logged-in with the state after logging out, including e.g., identifying flaws in session invalidation.

4.8 Extending Shepherd

As Shepherd provides functionality which is needed to explore post-login areas, it can be extended to widen its reach and scope. In the following, we present two extensions to Shepherd, that serve this purpose: logging in via a SSO provider and logging out.

4.8.1 Proof-of-Concept Facebook extension

In addition to domain-specific credentials, we build a Proof-of-Concept (PoC) to support single sign-on credentials. The main benefit of using SSO credentials is that this avoids the need to acquire a large set of valid credentials. On the face of it, it would seem that adapting Shepherd's core functions to login with SSO would be straightforward. In practice, we encountered difficulties doing this. As a result, we designed a further set of functions that allow Shepherd to handle the specific nature of logging in with SSO intermediaries. The following focuses on deviations from the domain-specific login design.

Searching for SSO login buttons. SSO login areas are usually activated by triggering an interactive element. Hence, the login area cannot be identified by searching for standard login elements such as a field for username or a link labelled 'login'. Rather, SSO-specific elements must be identified. Unfortunately, these elements are not standardised between SSO intermediaries and often differ between websites, even for one intermediary. An additional challenge is that some SSO intermediaries also provide social media features, such as a 'like' or 'share' button. Distinguishing such elements from the sought-for login elements is difficult. Similar to Zhou and Evans [ZE14], we address this by filtering interactive elements based on a set of keywords. These keywords are also specific to the SSO intermediary, which poses an additional challenge for supporting multiple intermediaries. We also encountered that in a case of insufficient filtering, these elements can lead to false positives as these produce similar login dialogues. We avoided these by using URL whitelists.

Submitting and verification. A hurdle is that after logging in with single sign-on, the website requests that the user fills in an enrolment form of some kind. Access to other parts of the site is blocked until this form is filled in. This behaviour does not always occur, but frequently enough that it affects the success rate. While that presents

an obstacle to achieving the full flexibility that domain-specific credentials offer, it does not impede gauging the usefulness of single sign-on for enhancing the coverage of Shepherd. For that, forms do not have to be filled in. Unlike domain-specific credentials, SSO credentials can hold additional details (e.g., real name or associated phone number), which can be used in future versions for improved verification procedures or to finish enrolment.

Implementation and performance. We developed an initial extension to Shepherd's core functions to login using Facebook's SSO service. In our testing set of 50 sites containing SSO logins for Facebook, it was able to identify login areas on 41 sites. Shepherd missed login areas on 7 sites, while two scans failed. Once Shepherd has found an SSO login, it processes the login as a regular login, but with the supplied Facebook credentials. The current identification process for Facebook logins is based on domain filtering (of URLs in the visited page). While this proved to be effective in practice, this does imply a certain amount of fine-tuning is needed to support additional SSO providers.

Due to the above mentioned difficulties in identifying the correct elements for SSO logins, the extension is significantly slower than regular Shepherd. The extension is able to scan about 3,000 sites per machine per day.

4.8.2 Validation of the Single Sign-On extension

We scanned the Alexa Top 10,000 sites using single sign-on credentials to validate the extension. We remark here that these sites do not all support single sign-on with Facebook credentials. This is intentional, as Shepherd should be able to scan any site. The scan was carried out with two machines, each using its own Facebook account, created newly for this purpose. We further divided the target domains into four equal sets, so that the results can be examined between scanning these sets. The first machine was used to scan the first three sets, while the second machine scanned only the last set. The Facebook account for the first machine was blocked at a certain point while scanning the second set, due to posting inappropriate content. Shepherd caused this by clicking share buttons on visited sites (of an adult nature). We adjusted the extension to address this by blacklisting certain types of Facebook URLs. After recovering the blocked account and scanning the third set, we found 55 shared posts on the account (each of which must be due to a successful login). Shepherd misclassified these as logins. The second Facebook account was also blocked, this time due to 'suspicious behaviour'. Recovery of this account was more involved and therefore omitted. Unlike the other three sets, we thus could not verify Shepherd's results for this set in the Facebook account.

As shown in Table 4.4, Shepherd thought it recognised Facebook-based SSO logins on around 20% of the Alexa Top 10K. While Shepherd was able to submit credentials to 93.1% of these, verification only succeeded on 20% of the *submitted* sites. It could be that Shepherd's default verification process is unsuitable for SSO logins, or, perhaps, often an additional registration form appeared and full site access was not yet granted. Shepherd set the percentage of sites on which it believed to have found Facebook Login at 20%, and the percentage of sites where it successfully verified login at about 4%. Logging in with Facebook leaves traces in the permission settings of

Table 4.4: Performance of the Single Sign-On scan

	# sites	out of	
total	10,000	–	
sites reached	8,829	10,000	(88.3%)
SSO login found	2,057	8,829	(23.3%)
submitted	1,915	2,057	(93.1%)
verified	383	1,915	(20.0%)
auth cookies found	330	383	(86.2%)

a user account. We checked this setting four our first account and found 664 apps with specific permissions. To compare these numbers, we looked for reliable numbers on the adoption of Facebook Login. The reported rates we encountered predicted significantly lower adoption. Nevertheless, we believe that the SSO login detection algorithm can be further improved to reduce false positives.

4.8.3 Logout Automation

We leverage the similarities between the logout process and the login process, which is already supported by Shepherd. In particular, our extension to log out follows similar steps, executed after a successful login.

The first step is to visit potential pages of interests. For our extension, we choose the page reached after logging in (likely a profile page) and the site landing page. Note that a well-designed website facilitates logout buttons on any page, after logging in. Second, we identify candidates for logout interaction elements. To this group belong elements that offer click functionality and contain keywords related to logging out. To determine if an element is clickable, Shepherd scans elements for attached event listeners, element tags (e.g., buttons, anchors etc.), and common properties of clickable elements. The third step is to define the order of elements to be triggered. For that, we rely on the distance of an element from the upper right-hand corner of the page. We noticed this aspect as a common property of logout elements during the development of our extension. This practice has also been shown to be successful for identifying login buttons in previous work [ZE14]. Fifth, Shepherd triggers these elements first by opening URLs from anchor elements, and then by performing mouse clicks. The final action is the verification of successful logout. For a verification, Shepherd visits the same page used to verify success of login, and checks whether login verification *fails*, i.e., the signals used to detect a failed login are used to detect a successful logout. More specifically, Shepherd uses the same information from the login phase to check whether the existence of login forms, logout elements, password fields and account information on the page has changed.

4.8.4 Validation of the logout process

We ran our logout extension on a set of 6,124 sites, where Shepherd successfully logged in. We found that logging out from existing sites is surprisingly difficult: we only managed to automate the logout process on 3,302 sites, which is 54% of the sites where we successfully logged in. We manually investigated causes for failing

logout. This revealed several causes. First of all, paths to logout elements vary more in labelling than for login elements. Some examples include logout, account, settings, profile, the actual username, "my" + the site's name, etc. Exacerbating this, some websites hide the actual logout interaction element in overlay menus. That is, there are websites that only inject logout interaction elements into the DOM when the corresponding menu is activated. Identifying and triggering such menus is much more challenging, as these vary in appearance and implementation. Another cause we found is related to banned accounts. For these sites, logging in succeeds, but any interactive element in the post-login phase is blocked, including, interestingly enough, the ability to log out. This problem is related to the used credentials, and cannot be resolved in the automation process. A third cause, seen in a small number of sites, comes from confirmation requests when triggering a logout. Integrating handling of logout dialogues is left as future work.

4.9 Conclusions

Many previous works have studied the Web. Most of these were limited to the public areas of websites. This implies that post-login aspects were hidden for such studies or could not be measured at scale. Research that attempted to address the post-login world, mostly fell back on manual intervention, to avoid the many challenges with an automatic approach [TDK11]. Only three previous studies had access to post-login areas of larger set of websites. All these used means to automate logging in, but were bound to certain type of logins. In this work, we took a generic approach to login on websites. For that, we designed and developed Shepherd, a tool that enables post-login measurements of unknown websites. As login processes are very diverse, automated logins cannot achieve full coverage. Moreover, the variety in login processes implies many design challenges. Shepherd accounts for this by several failure modes. The study conducted with Shepherd shows that large-scale automated login is feasible. Previous best efforts using a semi-automated approach [MFK16] managed 149 sites, while automated approaches reached 912 sites [ZE14] using SSO credentials.

In contrast, Shepherd can use either domain-specific or SSO credentials. In a case study with domain-specific credentials, Shepherd achieved 7,113 successful logins, an order of magnitude beyond previous best effort at logging in automatically. In a limited experiment with Facebook SSO credentials, Shepherd achieved 383 logins.

Future work. We are re-designing Shepherd's SSO component to use a more generic approach for SSO logins, which will lead to supporting more single sign-on frameworks and make the SSO-related procedures more robust. Given that there are various SSO providers commonly used in non-Western countries, this enables various types of detailed comparison studies between countries. Secondly, we are planning to extend previous studies of cookie security for post-login cookies. Related to this, integrating Shepherd-alike capabilities into a privacy measurement framework such as OpenWPM would allow to study whether logged-in users gain or lose privacy compared to anonymous visitors. Similarly, a combination with an existing security scanner could allow remote security scans (e.g., SQL injection, XSS, or CSRF) in the members-only area of websites.

Chapter 5

Multi-View Data Acquisition

Online shopping has simplified the collection of shoppers' personal data. Vendors potentially use this data to serve individual prices per customer. Several academic works have investigated how device or user characteristics may influence prices online. However, online vendors typically offer different stores to sell their items, such as desktop sites, mobile sites, country-specific sites, etc. Online rumours and news media reports persist that item prices between such views differ. To date, no systematic method for analysing this question has been put forth.

In this chapter, we devise an approach to investigate store-based price differentiation based on three pillars: a framework that can perform cross-store data acquisition synchronously, a method to perform cross-store item matching, and constraints to limit client-side noise factors. We test our approach in an initial case study to investigate store effects on flight pricing. We gather pricing data for 824 flights from 15 stores (incl. desktop sites, mobile apps, and mobile sites) over a 38-day period. Our experiment shows that price differences occur frequently. Moreover, even in a limited run, we find strong indications of store-specific pricing for specific vendors. We conclude that (i) a more extensive study into store-based price differentiation is needed to gauge this effect better; (ii) future research in this general domain should take store-based differences into account in their study design.

This chapter is based on the following publication:

Are some prices more equal than others? Evaluating store-based price differentiation. Hugo Jonker, Stefan Karsch, Benjamin Krumnow, and Godfried Meesters. *In Proc. 5th NDSS Workshop on Measurements, Attacks, and Defenses for the Web (MADWEB'23)*, DOI: 10.14722/madweb.2023.23011, 2023, [MADWeb23].

5.1 Introduction

Shopping is a basic fact of life that contains an interesting adversarial relation: the shopper wants to pay as little as possible, while the vendor wants to sell for as much as possible. In brick-and-mortar stores, either side has limited access to information to improve their side of this bargaining process. This changes radically for online shopping: customers can trivially look up prices of competitors, while vendors can leverage assorted technical measures to glean more information about their customers. This allows vendors to tailor their prices on the fly. With respect to this, we distinguish, as is common, between price differentiation and price discrimination. Price differentiation occurs when the same item is priced differently in another situation. Price discrimination occurs when this difference can be attributed to differences in user attributes between those two situations. Thus, all price discrimination is a form of price differentiation, but not vice versa.

Price discrimination has been the subject of various academic studies. In 2012, Mikians et al. [MGE+12] found indications of price discrimination using automated scraping. Since then, studies investigated the effect of user attributes (amongst others, device fingerprinting [HTW+18], user profiling [HSL+14]), explored different methods to collect data (such as Amazon Turk [HSL+14], crowd-sourcing [MGE+13; ISS+17]), and investigated the occurrence of price discrimination in different markets (e.g., airline tickets [VNB+14], rental cars [HTW+18]).

Less attention from the academic community has been devoted to group-targeted pricing. Several cases have occurred in real life, such as ZIP-code-driven pricing of homework tutoring in the US [AML15], or price differences between mobile apps and desktop sites as reported by German travel magazine *Clever Reisen* [Kau17]. Clever Reisen manually checked trip prices once, using up to four different stores (mobile and desktop). They found that prices can vary up to 8% between a company's mobile app and its website.

This informal investigation of price differences between two stores of the same vendor poses an especially interesting case: it is highly relevant to online shopping and could be systematically investigated from a user point-of-view, without access to the vendor's internal processes. Moreover, it is quite common for vendors to provide multiple online stores for their items, such as per-country stores (e.g., amazon.de, amazon.fr) or provide multiple versions of the same store (e.g., desktop/mobile site/app). While such stores cater to specific niches, the vendor could also choose to vary pricing between them. To the best of our knowledge, there has not yet been a systematic investigation into whether price differences occur between different stores of the same underlying vendor. In this chapter, we propose a method to systematically investigate store-based price differentiation, provide a proof-of-concept implementation and execute a limited validation test of the proof-of-concept against vendors of flights. For all vendors, we compare their app vs their German desktop site and their French site vs their German site. In addition, for one vendor, we compare its app vs its German desktop website vs its German mobile site. Even in this limited validation test, we find strong indications of store-specific pricing.

Contributions. We develop a device-independent approach to simultaneous data extraction. The approach relies on using dedicated machines to perform the data

extraction, either by accessing the web store themselves using Puppeteer or by interacting with a device (e.g., an app on a mobile phone) to do so using a dedicated extractor (for mobile phones, UI Automater and Appium). Data from apps and mobile sites are collected by using smartphones. We test our proof-of-concept implementation by collecting data on a handful of flights from 15 stores of 5 vendors over a period of 38 days. Our analysis of the collected data strongly suggests that store-based pricing occurs for more than half of the investigated vendors.

Ethical considerations. Our case study required visiting multiple stores by one company synchronously. This course of action can have a negative impact. First, booking systems for flights and accommodation tend to make a tentative reservation of items during the booking process to ensure the availability of the item offered to the client. During this time, the item is not available for other customers. To avoid this, we only collected data from search pages and never entered further into the booking process. To the best of our knowledge, bare item searching does not affect item availability.

Second, our queries could potentially impact prices shown to other subsequent visitors. The low number of queries per day produced by our bots should have negligible influence given the popularity and, therefore, large user base of the investigated services.

Third, we seek to minimise the impact on the services. Each comparison involves simultaneous data acquisition across all involved stores. We only collect data for one comparison at a time to avoid hammering the various stores. That is, data collection for the multiple comparisons is triggered sequentially. In addition, we do not issue repeat requests in case of a failure.

Outline. In this chapter's remaining sections, we present our design and implementation of a price comparison framework that accounts for different device classes (Section 5.2). We deploy to our framework to create a data set with flight prices from six vendors (Section 5.3). We use this data set to identify price differences (Section 5.4). Finally, we discuss the limitations (Section 5.5) and reflect our findings (Section 5.6).

Availability. Our Proof-of-Concept implementation, as well as our data set, are available for download [ART-MADWeb23].

5.2 Design of a price comparison framework

Studying prices on multiple platforms (e.g., mobile apps, desktop websites, etc.) requires two technical components: a data collection part, which retrieves item data from the studied vendors, and an item equivalence determination procedure, which determines whether two collected pieces of data concern the same item. An important goal of data collection is to eliminate or mitigate any price influences that are not subject to study. The purpose of item equivalence is to ensure that only equivalent items are compared, that is, to prevent the proverbial comparing of apples to oranges.

First, which items should be considered equivalent depends on the underlying study. For example, a study comparing prices of round fruits per weight would indeed compare prices of apples to prices of oranges. A study comparing prices of cultivars of apples would distinguish between golden delicious apples, braeburn apples, granny smith apples, etc., whilst considering small and large apples of the same cultivar equivalent. Note that due to EU law,[1] items must be purchasable for the price on display. Thus, additional fees can only concern extras; hiding necessary fees is not allowed. This implies that, for vendors selling to EU citizens, items can be compared once the vendor store displays a price. As item equivalence depends on the specific study under consideration, we detail our interpretation of item equivalence in the discussion of the validation experiment (Section 5.3).

Secondly, the data collector needs to reduce the effect of confounding factors. Here, the fact that this framework studies pricing on different stores comes into play. Various aspects may affect item price beyond the store used to view the item. Care must be taken to mitigate their impact, preferably by eliminating that completely. Alternatively, by ensuring their impact is constant or can be filtered out in another fashion.

5.2.1 Data acquisition methodology

To acquire item prices, we must collect data from each vendor. This can be done by automated tooling or manually. Both automated tooling and manual data acquisition require upkeep. Manual data acquisition is labour-intense, which we could overcome by using crowd-sourcing. Previous studies have shown that crowd-sourcing using a browser plugin eliminates the limitations of maintaining scrapers and allows targeting a broad number of stores [MGE+13; HSL+14; ISS+17]. However, this advantage does not hold when considering multiple device models (with different screens and resolutions) and classes (mobiles, desktops, tablets). This leads to a wide variety of vendor app/site layouts, which a crowd-sourcing solution should all encompass. Even more damning is that crowd-sourcing relies on individuals' devices. The vendor could employ price discrimination, which is a confounding factor for studying store-based price differentiation.

Nevertheless, automated scrapers are not perfect, either. As recent studies have shown [ESORICS19; JSS+21; CoNEXT22; CLB+22], desktop scrapers tend to be recognisable and recognised scrapers receive different results than regular user browsers. Therefore, measures must be taken to mitigate bot detection. The same applies to mobile scraping using emulated devices [YY20]. Previous price discrimination studies on mobile devices approached data acquisition by using modified desktop browsers with fitted userAgent strings. However, state-of-the-art bot detection companies use browser fingerprinting techniques [BMP+16; ESORICS19; VRR+20] and input from client hardware, such as mobile sensors [DAB+18], to distinguish bots from real human users. Therefore, to ensure that these aspects do not foul up mobile data acquisition, native devices should be used for the data collection.

Another approach to acquiring data pertaining to mobile phones is to gather pricing data directly from a vendor's backend for mobile users [CMW16]. This can be

[1] `https://europa.eu/youreurope/citizens/consumers/shopping/pricing-payments/index_en.htm`

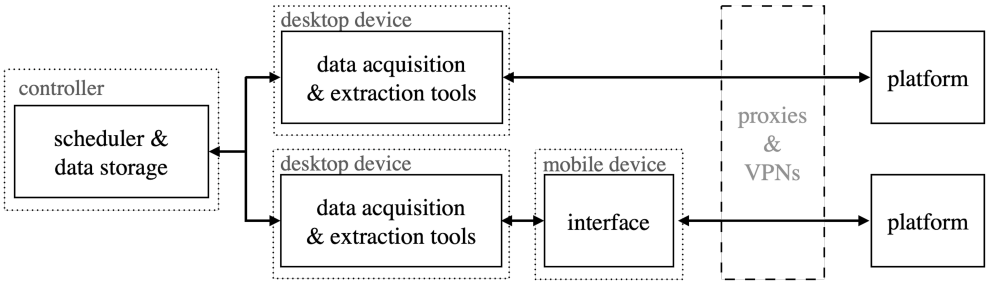

Figure 5.1: Framework design

done in two ways, by scripting user interface (UI) interaction on the client devices or by faking client interaction on the network level. Automation frameworks, such as Selenium, Puppeteer or Appium, offer APIs to script human interaction. Much like website scrapers, each scraper for a store requires an individual script. In addition, updates to the store's user interface may break scripts and, therefore, the data collection process – again, similar to website scraping. Faking client interaction on the network level has the advantage of skipping user interface aspects. However, network traffic is usually encrypted. While this is not a show-stopper, the process of bypassing encryption becomes cumbersome when the app uses key pinning. Even after circumventing encryption, we found that mobile apps typically connect to tens of different API endpoints. Since it is unclear what calls are sufficient to mimic an app, this makes faithful data acquisition via mimicking apps particularly hard. Thus, while mimicking apps can be less error-prone, initial development is far less straightforward than using scrapers, especially when targeting multiple vendors (i.e., unrelated stores/APIs). Moreover, by using UI interaction on client devices, maintaining data extractors on mobiles remains in the same category as desktop extractors: monitoring UI changes. Therefore, we make the design choice to script UI interaction for mobile data extraction.

5.2.2 Handling confounding factors

As stated, the goal of a framework is to eliminate or mitigate confounding factors. We broadly categorised previously reported aspects into four main areas: client-side aspects, timing aspects, contextual constraints, and pricing errors.

Client-side aspects. Client-side aspects include effects of browser fingerprinting, client profiling, location, and so on (see Section 3.1.3). These are necessarily present (even an empty profile is a profile); thus, their effects cannot be eliminated. A price measurement framework should keep client-side aspects constant as far as possible to limit this effect. That is: ensure the same browser fingerprint, start from the same user profile, use the same IP address, etc.

Timing. Progression of time affects prices, e.g., expiring items or tickets for events on specific dates. Vissers et al. [VNB+14] established that prices can already differ within a one-minute window. Moreover, increased interest may affect price –one

store's data collection may impact prices in another store. Due to network effects, it is impossible to ensure that queries for different stores arrive simultaneously. We point out that the goal of a framework is not to measure the digital world but to measure the human world. Therefore, we take the position that best-effort synchronisation is sufficient – if pricing is then still affected, a regular user would also encounter prices affected by random network delays.

Contextual constraints. Contextual constraints such as increasing energy prices, tax differences, natural disasters, etc., may affect prices. These should affect stores in the same context equally. Considering stores in different contexts (e.g., country-specific sites), the difference should be constant across all items. However, context can change over time (e.g., tax changes). A price measurement framework should collect data on multiple items over time to account for contextual effects on pricing.

Pricing errors. Pricing errors such as delays in propagating price updates or decimal point errors can cause different prices between stores. Vendors work to catch price errors. Thus, we can eliminate such effects by gathering item prices over an extended period.

5.2.3 Framework design and Proof-of-Concept implementation

Based on the problem analysis above, our system must be able to synchronise heterogeneous scrapers distributed over multiple devices. Further, it must support data collection on mobile devices and aggregate collected data into a single repository. This leads to the design depicted in Figure 5.1.

The central component is the scheduler that enforces the needed synchronisation. It schedules jobs to orchestrate other components in the system, supplies the information (search query input) to run scrapers (*extraction tools*), and manages a central data repository to persistently store results collected by devices. When a device retrieves tasks to query items from stores, it initiates a scraper. In our case, a *scraper* is an automated store-specific data-extracting client. It interacts with one store to request items. In the case of a mobile store, the design supports using an emulator as well as interfacing with an actual mobile device. In the latter setup, the mobile device acts as an interface for the scraper. Session initiation and communication with the store thus happen on the mobile device. Depending on the implementation of the system and target of the study, an external VPN server or proxy can be used to control outgoing IP addresses.

We created a Proof-of-Concept implementation to validate the design from the section above (see [Mee21] for a full, exhaustive description). An overview of our implementation is shown in Figure 5.2. The scheduler, *consumers/producers* and scrapers are written in JavaScript and run in a Node.js environment. To facilitate communication between the scheduler and other devices, we use the BullMQ framework,[2] a message queue system based on Redis.[3] We use one queue for desktop scrapers and a separate queue for mobile device scrapers. Finally, we use one queue for collecting data from

[2] `https://github.com/OptimalBits/bull/tree/develop/docs`
[3] `https://redis.io/`

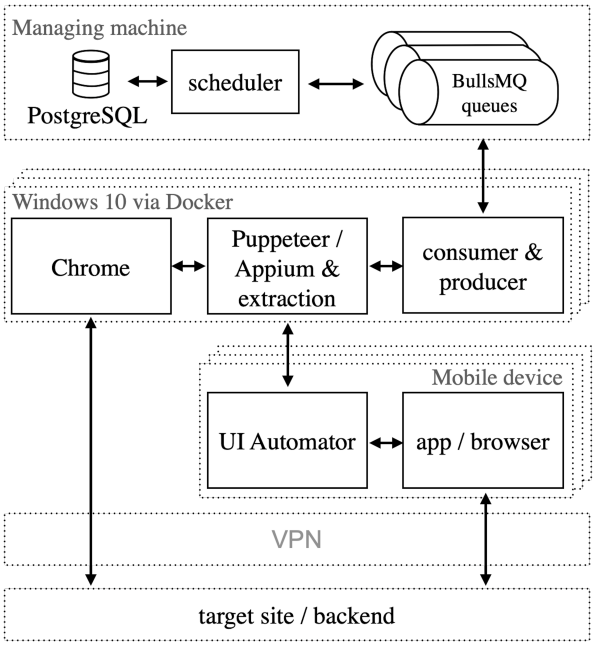

Figure 5.2: Proof-of-Concept implementation

all scrapers. The scheduler stores incoming results in a PostgreSQL database. We added a command line interface to control the scheduler and to push new tasks into a queue. We use this interface via cron jobs use to automatically push tasks to the workers.

The connection between consumers/producers and their scrapers is facilitated via interfaces in TypeScript. Every scraper must implement this interface, specifically, the methods start, stop, fill search, submit search, store results, and take screenshots. Website scrapers use Puppeteer[4] with the stealth plugin[5] for Puppeteer. For mobile scrapers, we connected the Android device via the USB port. To control the mobile device's interface from the scraper device, we set up an Appium[6] server on the scraper device and UI Automator on the mobile device. Appium then uses the UI Automator to extract and submit data and to perform UI tasks.

5.2.4 Client synchronisation

A data collection process involves multiple (at least two) scrapers that execute the following steps:

1. initialise session,

2. fill in search form,

[4] `https://github.com/puppeteer/puppeteer`
[5] `https://www.npmjs.com/package/puppeteer-extra-plugin-stealth`
[6] `https://appium.io/`

3. submit search form,
4. retrieve & parse items from result page, and
5. send parsed data to the scheduler.

The execution time of each step would vary on a single device between executions already. Our design allows for heterogeneous device classes, which exacerbates this problem. Our PoC implementation addresses this (see Section 5.2.2) by adding two synchronisation barriers; before step 2 and before step 3.

As soon as a device is ready for data acquisition, it retrieves a new task from the scheduler and initiates the corresponding scraper (step 1). The synchronisation barrier ensures that the filling in of the search forms is started simultaneously across all scrapers. We synchronise scrapers a second time at the beginning of the data acquisition phase (step 3). That is, we synchronise the submission of the search query. Note that due to the centralised design, we approximate simultaneous requests by triggering all scrapers to submit the query simultaneously. Nevertheless, implementation details and run time environments of the individual consumers, scrapers, and target's backend will introduce small timing variations.

5.3 Experiment: investigating flight pricing

In order to validate the design, we execute a modest experiment to investigate store-based price differentiation in flight tickets. We choose to focus on airline tickets, as these often are available via multiple stores (app, desktop site, mobile site, reseller). Moreover, while Vissers et al. [VNB+14] have not found any evidence of price discrimination in this domain, *Clever Reisen* [Kau17] claims to have witnessed price differentiation here. Besides validating the design's viability, our experiment's primary goal is to determine whether a large-scale investigation is warranted. To that end, we run our experiment over a period of 38 days for five popular[7] travel companies that offer flight tickets.

5.3.1 Data acquisition

To collect data, we select two airlines and three travel agencies. We conducted our experiment in May and June 2021, using all these companies to search for airline tickets. As input data, we consider European cities and pick random dates between July and August 2021. Further, we use corresponding options within search masks to query for one-way flights only. We left other options in their default setting. Before running our experiment, we manually checked that tickets were offered under equal conditions. For example, whether one store contains additional services compared to another. We verified this by installing the app or visiting the website to review the default setup. Our check revealed no differences between stores (see Appendix A).

For each company, we create scrapers to automatically query items via the mobile app, the mobile store, the German desktop website (*.de) and the French desktop website (*.fr). As we compare items retrieved from mobile websites exclusively for

[7] With the exception of airfrance.de (Top 240K) and kayak.de (Top 140K), all domains are listed in the Tranco Top 100K [LVT+19] (ID K2Z6W).

one company, our experiment covers 15 stores in total. With these store scrapers, we construct comparisons by running at least two different scrapers synchronously. For comparisons that include mobile apps, we use German localisation. To do so, we visit the company's .de domain with our website scrapers. For the mobile scraper, we set the system language to German and downloaded the corresponding localised mobile application and set the phone's GPS location to a place in Germany, co-located with the town of the IP-address origin.

For each two-way comparison, we collect data thrice daily between 25 May and 2 July 2021, for a maximum of 114 data points for comparison. Note that to be used in a comparison requires all involved scrapers to be successful, which underlines the fragility of the data acquisition process. Scraper failure, a regular occurrence, happens due to frequent layout changes, changes in flights on offer, as well as network hiccups. This necessitated frequent scraper updates. We also run one three-way comparison (app/web/mobile). Since this comparison requires three simultaneous successful scrapes for one comparison point, we foresee a dearth of comparison points. To mitigate this, we run these scrapers longer until we collect over 70 comparison points. They ran from 25 May to 5 July (maximum of 123 comparison points). We collect items shown on the result page without following the booking process to avoid blocking items for other store users.

Finally, during each run, we took screenshots of all result pages and recorded HTTP responses containing JSON-formatted data. A random selection of these was then manually compared with the collected data. This verification step found no errors in the automatically collected data.

5.3.2 Determining item equivalence

We must identify equivalent items from two (or more) stores to compare items. We conduct multiple steps to determine whether items are equivalent: attribute normalisation, relevant attributes, and robustness against minor attribute changes. We point out that the choices made here significantly affect what items are considered equivalent and, thereby, the nature of the study. In our case, we aim to determine the price of a trip between two airports. If departure and arrival airports are equal, and departure and arrival times are identical, we consider the items equivalent. This assumes that no two flights would leave from one airport *and* arrive simultaneously at the same destination airport. This is not true for codeshare flights, but we consider these as the same flight.

To determine item equivalence for our setting, we first normalise attribute values ('Heathrow', 'London – Heathrow', and 'LHR' → `LHR`). More specifically, we check all collected attribute values and manually map these to normalised values. For numeric attributes, we establish a standard format and map all values to the standard format. Secondly, we decide on the set of attributes to include for determining item equivalence. We chose the set {date, time, departure airport, arrival airport}. Omitting attributes such as flight number and airline helps to unify codeshare flights. Lastly, shifts in departure/landing time occur. These constitute different items, which we easily identified and manually matched with their originals throughout the data set. This makes the data set robust against minor changes in these attributes.

Table 5.1: Overview of flight data set

company	scrapers	date		orig–dest	#comps.	#trips
Air France	app/web	July	01	FRA–CDG	73	2 (2)
		August	01	FRA–CDG	72	2 (2)
		August	09	VIE–AMS	72	3 (3)
	.fr/.de	July	01	FRA–CDG	68	2 (2)
Eurowings	app/web	July	11	AMS–HAM	8	1 (1)
		August	12	CGN–LON	7	3 (0)
	.fr/.de	July	11	AMS–HAM	11	1 (1)
Expedia	app/web	July	01	BRU–AMS	29	24 (3)
		August	10	AMS–ARN	95	7 (6)
		August	18	OPO–BRU	101	20 (6)
	.fr/.de	July	01	BRU–AMS	74	5 (4)
		August	01	BRU–AMS	83	4 (4)
KAYAK	app/web/mob.	August	18	OPO–BRU	71	154 (4)
	app/web	August	07	MAD–FCO	109	247 (13)
		August	13	BER–BCN	101	273 (6)
Opodo	app/web	July	01	FRA–CDG	99	9 (9)
		August	01	FRA–CDG	90	18 (7)
		August	23	CGN–PRG	103	16 (1)
		August	18	OPO–BRU	101	17 (4)
	.fr/.de	July	01	FRA–CDG	101	9 (9)

5.3.3 Resulting data set

Table 5.1 shows which vendors/stores were visited, describes the used input search queries, and describes the data points collected. The #trips column shows the total number of unique trips (modulo schedule updates) found. Each trip is one journey from departure to arrival airport, including any intermediate stops. In parenthesis, this column shows the number of equivalent trips found across multiple stores simultaneously. We only compare prices if we find identical items from different stores simultaneously at least once. In some cases, there are vast discrepancies between these numbers. We found three major effects for this: first, for vendors who return many trips, much fewer trips are visible on mobiles (both app and mobile site) than on the desktop without interacting. Second, some vendors offer multi-stage alternatives for direct flights, such as a bus transfer to an alternate departure airport or a train trip. These two effects explain most of the discrepancy. Lastly, some trips only occur once or a few times. We typically see these only on one store (possibly also due to the first effect). We include the number of trips found on more than one store as only these can be used to analyse store-related price differences. Finally, the #comps column counts runs where all involved scrapers successfully retrieved their result page. These are the data points that can be used for data analysis – the number of data points for each trip for which we collected data simultaneously from multiple stores. Note that these may not be uniformly distributed over the trips found on multiple stores.

5.4 Analysis

Our analysis aims to evaluate whether our approach can find any cases of store-based price differences. In addition, we want to attribute any found differences to likely

causes. Our analysis proceeds in three stages. First, we consider, for each vendor, the distributions of relative differences between stores (via boxplots). Second, we consider patterns in pricing via a heatmap. Third, we consider price differences for specific trips, plotting their price developments on both monitored stores.

5.4.1 Occurrence of store-based price differentiation

We collected sufficient data from AirFrance, Expedia, Kayak, and Opodo to proceed with our analysis. Figure 5.3 shows the distribution of relative price differences between their (.de) desktop site and other stores. We found some price differences for each store, though, for AirFrance and Expedia, these appear to be minor or incidental. In contrast, the distributions of Opodo and Kayak show apparent favouritism.

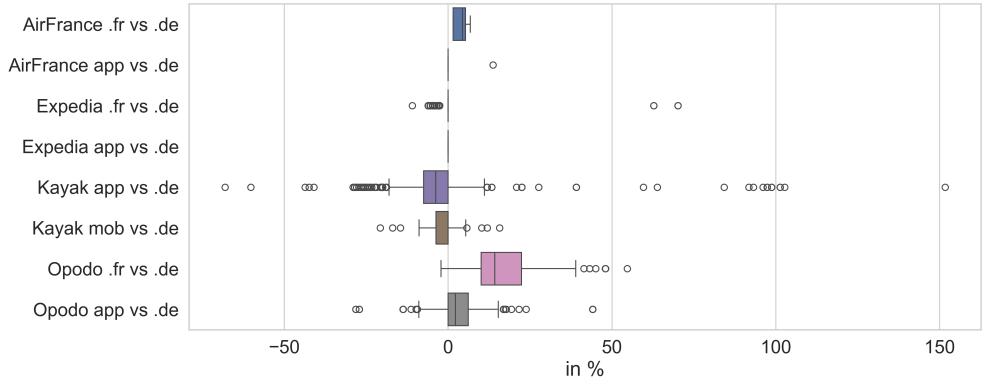

Figure 5.3: Relative difference vs .de desktop site

Opodo: French site almost always more expensive. Even our data set provides a limited window, for Opodo, we find that prices on the French website are typically substantially higher than on the German website (median: +14.2%, InterQuartile Range: +10.1%–+22.4%). Indeed, virtually always a lower price was offered on the German website. In Opodo's app, this effect is also present, but smaller (median: +2.25%, IQR: +0.01%–+6.15%). Moreover, while outliers occur in either direction, those favouring the German site over the app are quantitatively larger and occur more often. This is confirmed when looking at a single item, e.g., Figure 5.4: users of the French site pay, in this case, tens of euros more for the same ticket. On the 22^{nd} of June, this was even over €55. While lower prices do occasionally occur on the French site, on the whole, our data set suggests Opodo customers are better off using the German desktop site for booking.

Kayak: wild variations between stores. For Kayak, we find, on average, mobiles get better prices. Prices on all stores vary; in-app prices are mostly slightly better than on the desktop site (median: −4.3%, IQR: −8.0%–0.0%). However, wild outliers occur in either direction. Lucky bookers will be 68% cheaper; unlucky bookers will see more than double (over 150% higher) prices than on the desktop site. Kayak's

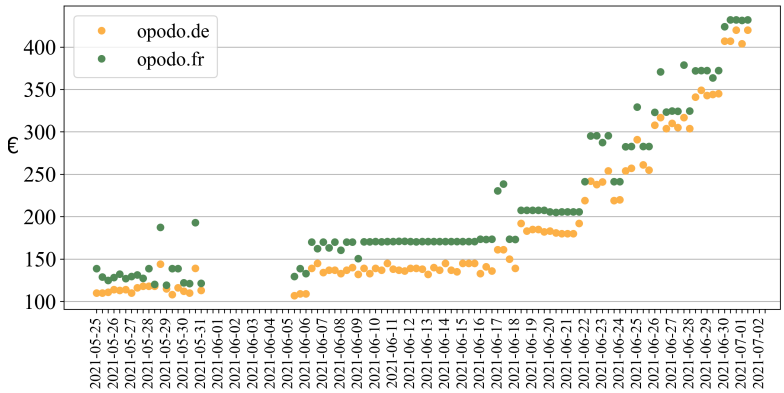

Figure 5.4: Opodo, .de vs. .fr, FRA – CDG

mobile site also offers better prices (median: −1.6%, IQR: −4.4%–0.0%). Outliers for the mobile site are less wild, ranging from −20% to +15.7%. On the whole, the best bet for low prices from Kayak is (by a small margin) their app. Its users might even get lucky with a very low price but should be wary of expensive outliers.

5.4.2 Pattern and outlier analysis

To evaluate whether there are patterns in the found price differences, we construct a heat map of these (Figure 5.5). We omit flights with less than 10% success in data acquisition. Each box shows the relative difference of an item's prices between two stores, with time of data acquisition on the x-axis. A lacking box means our scrapers acquired insufficient data; a light grey box means that both stores showed the same price. All other boxes denote price differences. For visual clarity, all boxes with data are marked with an edge. Extremes use the 50% colour to avoid letting outliers dominate the colour space.

Using relative percentage differences instead of total prices provides a compact view on the data set. At the same time, differences in higher-price trips do not overshadow differences in low-fare trips. On the downside, relative differences result into higher percentages for increases than for decreases.[8] Hence, this view can provide points of interest, but conclusions require further analysis. Below we discuss high-level observations from this heatmap and indicate which needs further inspection.

Cross-vendor observations. Any cross-vendor effect manifests as a vertical effect in the heatmap. This includes gaps in data acquisition. First, our data acquisition failed to get data from any app from the 5^{th} of June until the 8^{th} of June. This shows up in the heatmap as a vertical gap for all app comparisons. Secondly, other effects besides gaps in data acquisition also occur vertically. An interesting find is that from Figure 5.5, we can rule out taxes as a cause for the observed price differences between .de and .fr sites. If taxes caused price differences, this effect would show up in all

[8]For example, a change from €10 to €15 is an increase of 50%, while from €15 to €10 is a decrease of 33%.

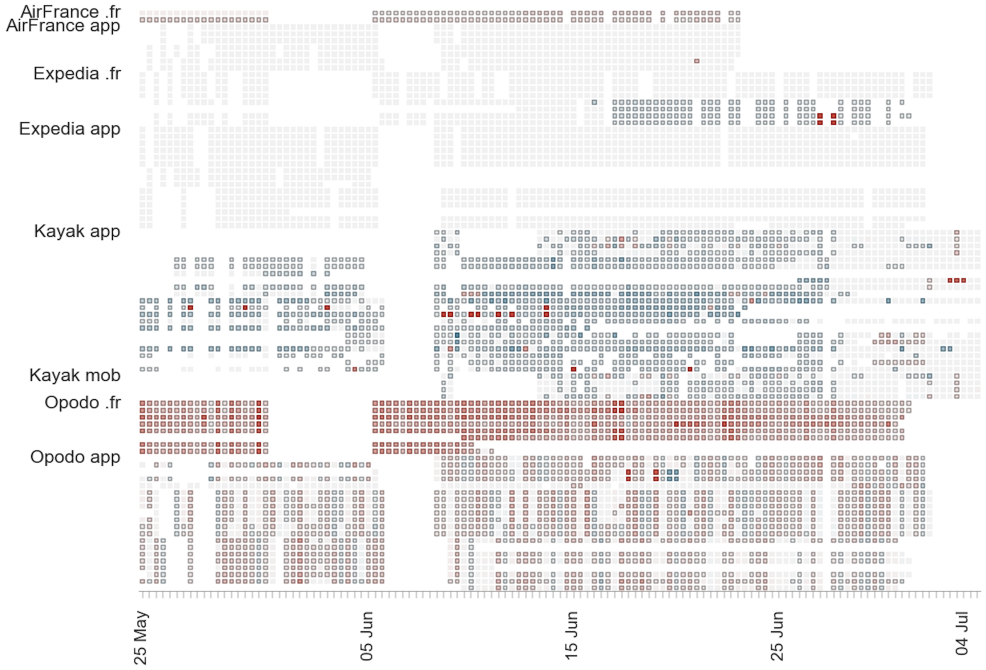

Figure 5.5: Overview of relative price differences (colours of extremes truncated to $+/-50\%$)

affected data points, translating to a vertical effect in the heatmap; no such effect is present. This is further strengthened by observing that most Expedia .fr comparisons show no price difference, except when the booking date is close to the departure date.

Vendor-specific observations. With respect to specific vendors, we observe several effects in Figure 5.5. First of all, we see that the AirFrance .fr site seems to have an almost-constant price increase over their .de site. This could relate to some type of fee. Secondly, we see a wave pattern for the Opodo app indicating frequent switches between higher and lower prices. We will consider both cases in more depth in the next section.

Perspective on outliers. The heat map also provides more details about recorded outliers. We note that outliers for Expedia (cf., Ex .fr in Figure 5.3) are grouped (i.e., on consecutive dates). We will investigate this case further to find the cause for this pattern below. For AirFrance .fr, AirFrance app, and Opodo app, the fraction of outliers is tiny. We ignore such single-point outliers, as these could be due to expected errors given the nature of our experiment (see Section 5.2.2). Instead, we focus on more consistent data.

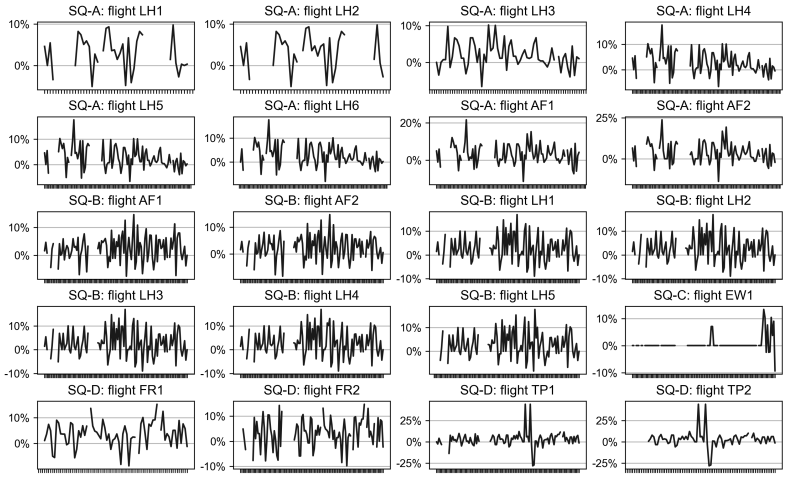

Figure 5.6: Opodo website vs. app, differences per item

5.4.3 Analysis of specific cases

Price differences flip frequently for Opodo. The wave pattern perceived in Opodo's part of the heat map spans over all four search queries, with the exception of one item. We checked that these price differences are significant (p-value$\leq$0.0001, Wilcoxon signed-rank test).[9] This raises the question of whether differences in item prices follow a predictable pattern. Figure 5.6 shows the relative difference between website and app prices for each item of each search query (SQ-A through SQ-D). Breaks in the plot occur where data is lacking. For each query with multiple items, we can find some items whose plots are similar, such as SQ-A: LH1 and LH2, SQ-B: all plots, SQ-D: TP1 and TP2. On the other hand, each of these also has dissimilar plots.

Note that none of these patterns re-occurs for items from different search queries. We only have four queries, none of which show the same pattern. Data on more queries would be needed to put any specific inference on a firm footing. Nevertheless, a common thread already evident is that most items show a high frequency of price swings – hence the wave. This suggests that, in general, checking prices on Opodo's other store can save around 10% of the item price.

Expedia: booking last minute? Try switching stores. In the heat map, some items on expedia.fr have lower prices than their .de counterpart in the last third of our observation window. Interestingly, the other four rows within this comparison category show no differences. One example with differences is depicted in more detail in Figure 5.7. Here, flight prices start to differ by a constant rate 15 days before departure (except for two outliers). Customers on expedia.fr pay €13 (initially: −6.1%) less than on expedia.de for these 'last-minute' tickets. The four search queries that show no differences in the heat map all have later departure dates (August 1); for these, we lack similar 'last-minute' data. As previously remarked, tax differences be-

[9] We use Wilcoxon signed-rank test, as our data set is not normally distributed.

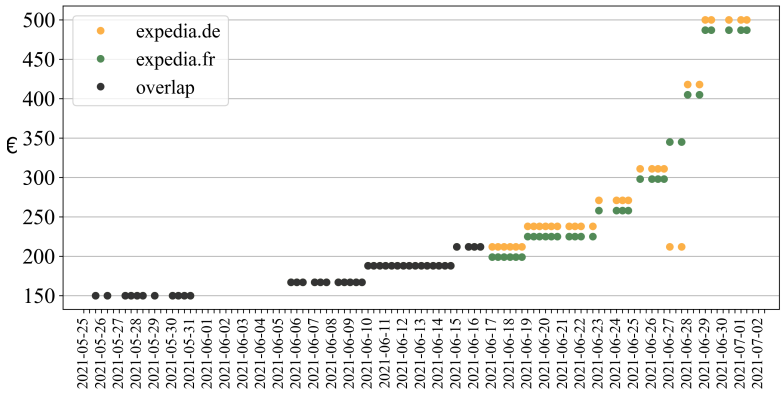

Figure 5.7: Expedia, .de vs. .fr, BRU – AMS. Data points under overlap denotes equal prices on both stores

tween countries would show up in many more points than observed, so these cannot explain the observed differences.

€1 or €5 higher prices on airfrance.fr. For two AirFrance flights, we have sufficient data to compare prices between .de and .fr sites. Both flights showed exactly two levels of price differences: €1 and €5. Interestingly, there is no consistent temporal component to this (see Figure 5.8). For both flights, only prices over €65 are €5 more expensive on the .fr sites. Unfortunately, our data set only contains sufficient data to analyse two flights; the data we have is tantalising and merits a follow-up study.

5.5 Limitations

The experiment conducted in this chapter has several limitations. First, our validation experiment only aims to show the framework's viability to support price differentiation studies. As such, we only extracted pricing data from a small number of stores of five European vendors of flights. The experiment's conclusions should not be extrapolated to other vendors, and, given its scale, one should be careful extrapolating our results to other (non-investigated) flights of the same vendors. Nevertheless, the experiment's results show that even a limited experiment with a proof-of-concept tool can already provide indications of (trends in) price differentiation.

Second, our experiment used blank profiles (reset for every scraping job). Note that this is a limitation of the experiment; the framework already supports using different profiles (or even browsers) simply by instrumenting a scraper with the desired characteristics as one of the desktop devices. With respect to how blank profiles affect data: although no link between browsing history and flight pricing has yet been found, browsing history is known to affect search results [MGE+12].

Third, our experimental setup only partially eliminates all possible confounding factors. For example, store-specific measures such as user tracking (e.g., [Eck10; LBB+20]), bot detection via fingerprinting [ESORICS19; VRR+20], behavioural metrics [IMC21], or A/B testing could still impact results. We argue that the impact of

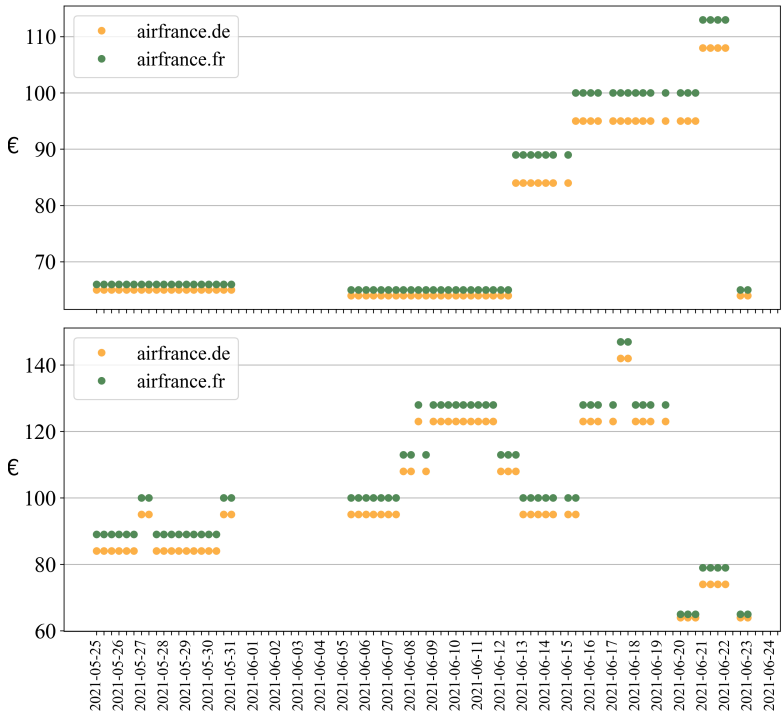

Figure 5.8: AirFrance, .de vs. .fr, FRA – CDG, AF1 top; AF2 bottom

any confounding measures, not specifically due to detecting the client as a bot, would similarly affect genuine clients starting from the same position as our experiment. Bot detection poses another potential source of confounding factors. Stores could employ bot detection to prevent or thwart unwanted resellers. Specifically, they could block bots, show bots different prices, or show fake or joke items. The last two measures can thwart bots and may even allow the store to (significantly) profit from automated resellers [Sch19]. Also, some of the observed differences could be caused by A/B testing. To perform A/B testing, a store puts users into buckets on a random basis; these users then retrieve prices according to their corresponding buckets. A/B testing is a method for temporary use that should not lead to large discrepancies between platforms over a period of 38 days. If not, such algorithms are likewise problematic as pricing strategies that do not use A/B testing.

Lastly, any measurement we took from a store produced data potentially used by this store for adjusting item prices. This phenomenon is known as the observer effect, as the act of observing the system may cause it to change. However, given that hundreds to thousands of users are active on an online store, our experiment should have limited influence on a vendor's pricing.

5.6 Conclusions

In this work, we set out to explore the feasibility of detecting store-based price differentiation. In contrast to previous price-differentiation studies, we focused on vendor-side differences in pricing (that is, store-based pricing) and conducted a fully automated study. Our experiment is based on a significantly larger amount of data than previous works in this area. Studying store-based price differentiation requires eliminating or mitigating confounding factors as well as gathering the same item from multiple stores. To this end, we designed a framework to enable cross-device automated data collection from different stores. We implemented a proof-of-concept that uses Puppeteer for automating desktop browsers and Appium plus UI Automator for automating interaction with mobile devices.

We test our approach in a study of 5 vendors of flights across 15 stores, gathering data for 38 days. We find evidence of price differentiation between stores for about half of the data set. For some vendors, this seems incidental. For others, the collected data is suggestive of deliberate price differentiation – sufficiently strong to warrant a more detailed, in-depth study. Lastly, for some vendors, price differences are fairly blatant. We found recurring price differences of about €50 between .de and .fr stores of Opodo. This starkly contrasts data from all other investigated .de and .fr sites, ruling out contextual factors such as tax differences. We also created comparisons between websites, mobile apps, and mobile sites. We find that Opodo's app slightly disadvantages users compared to their desktop site. Kayak's stores can differ substantially in price; on average, their mobile stores (app/mobile site) offer lower prices.

Future work. Our experiments show that a larger-scale study into store-based flight pricing is warranted. This can be done in various dimensions. One such way is to perform a deep dive into one vendor, gathering data on a significant fraction of their flights. This could e.g. be done for Opodo. Another large-scale investigation is to collect data from many more vendors to determine if there is an industry-wide trend with respect to store-based pricing. Do note that these studies have different challenges: a vendor-specific investigation must take extra measures to reduce the impact of its queries on pricing, while a multi-vendor study faces the challenge of maintaining a plethora of vendor-specific data extractors. Another interesting direction related to cross-vendor studies is to consider seller incentives. For example, in some cases, resellers stimulate offering lower prices to mobile devices than those offered to desktop clients [Boo23]. A future study could investigate to what extent such features are used across different vendors.

In our experiment, we studied flight prices. Store-based price differentiation can, of course, also occur for other types of items. Travel-industry-related rental items such as hotel bookings or car rentals make interesting candidates. They also allow for fairly easy item comparison. Moreover, these items also have limited availability and are time-sensitive, which is likely to induce vendors to update their pricing. From there, it is only a small step for vendors to consider tweaking prices for specific stores. Hannak et al. [HSL+14] reported finding lower prices for hotel bookings on Apple devices than on Android devices for one out of four vendors. Our framework facilitates performing such studies at a larger scale, for more vendors and across more stores.

In all of such price differentiation studies, data extraction remains a contentious point due to server-side changes. On a conceptual level, a follow-up study is to investigate how to make data extraction significantly more robust. Lastly, taking a step back, there have been various studies into price differentiation focusing on different aspects. Not all studies insulated their data collection from aspects found to be influencing pricing in other studies. What is sorely needed is a taxonomy of various potential influencing factors and a strategy for isolating them.

Chapter 6

Case Study: Session Security from Pre-login to Post-logout

Session management is a particularly delicate component of web applications, which might suffer from a range of severe security issues, including impersonation attacks. Unfortunately, the scope and significance of prior work on web session security in the wild are limited by the complexity of the attack surface and the challenges of automating the login process on existing websites. In this chapter, we fill this gap by proposing the first comprehensive, large-scale web session security measurement based on post-login *data. Our analysis is comprehensive in that it deals with all key aspects of web sessions, i.e., the login process, the logout process and the authentication cookie handling. Our automated approach analysed an extensive set of session management practices of over 6,000 sites where login was successful and authentication cookies could be automatically detected, uncovering a widespread adoption of insecure practices in the wild.*

This chapter is based on the following publication:

Measuring Web Session Security at Scale. Stefano Calzavara, Hugo Jonker, Benjamin Krumnow, and Alvise Rabitti. *In Journal of Computers & Security*, DOI: 10.1016/j.cose.2021.102472, 2021, [CoSe21].

6.1 Introduction

Web application security is a complex matter, with multiple facets and moving parts. A particularly delicate component of most web applications is *session management*, where a user operating a client (browser) authenticates at a web application to request access to security-sensitive functionality, e.g., a payment interface of an e-commerce website. Web sessions are normally established upon successful verification of valid access credentials (login) and implemented on top of *authentication cookies*. Unfortunately, despite their apparent simplicity, web sessions can suffer from a wide range of severe security flaws [CFS+17]. Insecure implementation practices in web sessions may even lead to impersonation attacks, where the attacker uses the victim's password or cookies to authenticate as the victim and get unconstrained access to their account.

Web security studies aim to better understand causes for insecure practices; they unveil faulty implementations and highlight misunderstood concepts [KB15; WSL+16]. However, web session security studies have been fairly limited so far. Analysing web session security requires authenticated access to web applications, which is a difficult process to automate (see Chapter 4). Thus, prior work on web session security reported on either (i) small-scale precise measurements involving a significant amount of manual effort [SPK16b; MFK16; CRB19], or (ii) large-scale measurements based on unauthenticated access to web applications, which miss valuable information, e.g., the login and logout processes [BCF+15]. The only notable exception is a recent paper, which analysed post-login web session security at scale, but only focused on session hijacking enabled by cookie theft [DIP20]. This means that prior web security studies are too small in terms of analysed sites [SPK16b; MFK16; CRB19], too imprecise because carried out without performing authentication [BCF+15] or too narrow because they only cover a limited set of web session security threats [DIP20]; we further discuss and compare against prior work in Section 6.8.

In the present chapter, we fill the gap in prior studies by presenting the first *comprehensive evaluation* of web session security that is based on *post-login data* collected through an automated *large-scale measurement*. Our analysis is comprehensive because it deals with all key aspects of web sessions, i.e., the login process, the logout process and the authentication cookie handling. Note that all these parts of the session management logic may be subject to vulnerabilities:

1. Web session security requires passwords to be protected against leakage over HTTP and to be reasonably hard to guess. If passwords are not appropriately protected against disclosure, impersonation attacks become trivial to perform.

2. Once a session is terminated by logging out, it should be invalidated at the server-side to ensure that authentication cookies are not valid beyond their intended expiration. Also, security-sensitive information stored at the client should be removed to minimize the risk of privacy leakage.

3. Insecure cookie configurations can fatally undermine web session security. For example, if authentication cookies are leaked in clear over HTTP, their theft may enable impersonation attempts (session hijacking).

We build our work on top of the Shepherd framework (see Section 4.2) for automated post-login studies, which we extend to mechanise the logout process and

include new traffic collection facilities. Our analysis is designed to be non-intrusive and ethical: we leverage existing access credentials of popular sites from the public BugMeNot[1] database and we check compliance with security best practices without actively mounting attacks when we might violate existing terms of services. Despite these necessary limitations, our analysis is valuable because it identifies widespread adoption of insecure session management practices in the wild. We arrive at this conclusion by analyzing data collected after authenticating to 6,124 top sites from the Tranco list [LVT+19]. More concretely, our study shows that the risk of impersonation attacks on the analysed sites is significant: for example, we identify 909 (15%) sites where impersonation might be enabled by an insecure implementation of the login process and 1,398 (23%) sites where impersonation might be enabled by the lack of confidentiality of authentication cookies. In addition, we identify a number of sites which implement the logout functionality insecurely: specifically, 469 (8%) sites do not terminate sessions at the server upon logout, while 230 (4%) sites do not remove security-sensitive information from the client after logout. All the vulnerabilities reported in the present article have been responsibly disclosed to the respective site operators.

Contributions. To sum up, we contribute as follows:

1. We use Shepherd (see Chapter 4) to create a data set of traffic and client-side storage related to all phases of session security: logging in, post-login, logging out. For this task, we extend to capture targeted parts of the HTTP traffic. This enables Shepherd to make use of its understanding of the login / logout processes during traffic collection and support further security analyses.

2. We review an extensive set of web session security threats, focusing on three different angles: login security, post-login security and logout security. For each threat, we identify automated testing techniques amenable for a large-scale security measurement in the wild.

3. We apply these testing techniques to data collected from 6,124 sites of the Tranco list [LVT+19] where Shepherd successfully logged in. We analyse the results to shed light on the current state of session security on those sites, detecting a widespread adoption of insecure practices.

Ethical considerations. The data collection process in this study relies on the Shepherd framework. Likewise, the ethical standards from Section 4.1 apply to this work.

In addition, our research uncovers vulnerabilities, which raises concerns regarding the creation of such a sensitive data set. We take two precautions to reduce potential negative effects that could be caused by our study. First, we deploy measures to protect the data set against unauthorised access (encrypt hard drives and limit access to responsible personnel). Second, most of the sites found to be vulnerable are not named in our report. An exception are site operators that we explicitly mention for illustrate real world cases (e.g., see Section 6.4.1). To reduce possible harm to these

[1] `http://bugmenot.com`

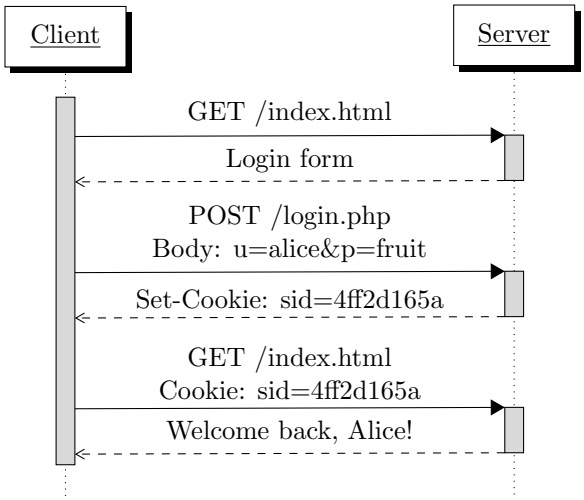

Figure 6.1: Example of web session

sites, we reported found flaws to site operators under the guidelines for responsible disclosure as stated by the OWASP foundation [Tea21].

Outline. In this chapter, we continue with providing background knowledge on session security (Section 6.2). Afterwards, we describe our experiment in detail and evaluate the representativeness our of data set (Section 6.3). For the analysis, we report on login (Section 6.4), post-login (Section 6.5), and logout security (Section 6.6). We further discuss the meaning of our findings for website security (Section 6.7) and we contrast them with related work (Section 6.8). This is followed by concluding remarks (Section 6.9).

6.2 Studying web session security

A web session is established when a user operating a client (normally a web browser) provides valid access credentials to a web application by the submission of a login form, which is sent to a remote endpoint (the form's *action*) for verification. Normally, upon a successful verification of the access credentials, the web application issues a set of *cookies* which authenticate the user on the following HTTP requests [KM00], e.g., because they store a unique session identifier bound to the user's identity. Such cookies are known as *authentication cookies*[2] and are automatically sent by the client to the web application which set them.

Figure 6.1 shows the typical establishment of a web session, where the user Alice first logs in with password "fruit" and then remains authenticated by presenting an authentication cookie sid, which uniquely identifies her session (4ff2d165a). Once

[2]This is interchangeable with the term *session cookies* in some other work. We avoid the use of the latter term, since it can also be used to denote those cookies which are deleted when the browser is closed.

Alice has finished interacting with the web application, she can log out and move back to an unauthenticated state (not shown in the figure). This makes her session identifier invalid for future accesses.

6.2.1 Threat Model

We audit the security of web sessions against the traditional threats posed by *web attackers* and *network attackers*, the standard attacker models of the web security literature [ABL+10], which have been commonly used in previous web session security studies, e.g., [CNH+13; ZJL+15; BCF+15; vAHS17; CFS+17; CFN+19; SRJ+19; DIP20]. A web attacker is an unprivileged web user who operates a browser and has control of a malicious website. A network attacker extends the capabilities of a web attacker with the ability to inspect and arbitrarily modify the content of the HTTP traffic exchanged between the client and the server, e.g., because the attacker has control of the WiFi access point used by the client and operates from a man-in-the-middle position. However, a network attacker cannot sniff or corrupt the content of HTTPS traffic, assuming the adoption of robust cryptography and the deployment of a trusted certificate on the server. In our analysis, we only focus on sites equipped with certificates signed by a trusted certification authority according to a major commercial browser (Google Chrome). We also assume perfect cryptography, in the sense that our analysis focuses on session security, not cryptographic security of HTTPS. Note that cryptographic weaknesses in HTTPS implementations are generally harder both to identify and to exploit in practice [CFN+19].

Finally, for the specific case of logout security, we also consider a *next user attacker*, who gains access to the client after the previous user has logged out of her session. This attacker covers often overlooked threats related to sharing devices, such as borrowing someone's computer or using an Internet cafe. The next user attacker has access to the same browser and resources as used by the victim. More specifically, a website that does not clean up client-side storage upon logging out leaves behind information in the form of cookies and localStorage items. Information in sessionStorage is safe, because sessionStorage is deleted when the user closes the corresponding browser tab.

6.2.2 Web Defences

There are common defences to improve session security. A few of them are explained in this section.

Cookie attributes and prefixes. To understand the security implications of cookies, it is important to review their semantics. By default, cookies are only attached to requests sent to the same host which set them. However, a host may also set cookies for a parent domain by means of the `Domain` attribute, as long as the parent domain does not occur in the Public Suffix List:[3] these cookies, called *domain cookies*, are shared across all the sub-domains of such domain. For instance, `a.foo.com` can set a cookie with the `Domain` attribute set to `foo.com`, which would also be sent to `b.foo.com`.

[3] Available at `https://publicsuffix.org/`

Cookies are normally shared across all protocols and ports. For instance, cookies set by a secure connection to `https://www.foo.com` are attached to insecure requests to `http://www.foo.com`, i.e., they can potentially be stolen by network sniffing. To improve their confidentiality guarantees, cookies can be marked with the `Secure` attribute, which instructs browsers to communicate such cookies only over HTTPS connections. Similarly, cookies can be shielded from JavaScript accesses by marking them with the `HttpOnly` attribute, which mitigates the dangers coming from script injection (XSS).

The lack of cookie isolation between protocols also implies that `http://www.example.com` can set cookies for `https://www.example.com`, i.e., cookies lack integrity against network attackers [ZJL+15]. To avoid this, cookies can make use of the security prefixes `__Secure-` and `__Host-`. Though the semantics of the two prefixes is different, both of them require the cookie to be set over HTTPS connections, thus providing cookie integrity.

HTTP Strict Transport Security (HSTS). HSTS is a security policy implemented in all modern browsers, which allows hosts to require browsers to communicate with them only over HTTPS. Specifically, HTTP requests to HSTS hosts are automatically upgraded to HTTPS by the browser before they are sent. This way, site operators can assume that HTTP is banned and reduce the attack surface. Note that HSTS provides better protection than a standard HTTPS deployment (without HSTS), because HTTP communication is entirely forbidden, hence network attackers cannot impersonate the (non-existing) HTTP version of the target site.

HSTS can be activated over HTTPS using the appropriate header, which must specify a `max-age` attribute expressing the duration of protection. Moreover, the header can set the `includeSubDomains` option, which extends the scope of HSTS to all subdomains. Rather than activating HSTS via headers, hosts may request to be included in the HSTS preload list of major web browsers,[4] so that HSTS is activated on them by default. HSTS can be deactivated by setting the `max-age` attribute to a non-positive value.

6.3 Data collection

We now provide details about our data collection, where we use Shepherd (see Chapter 4). We start by explaining our modifications and then describe the data collection process. Finally, we zoom in on our data set and analyse its characteristics.

6.3.1 Enhancing Shepherd with a network traffic recorder

The standard version of Shepherd provides access to a website's JavaScript, WebStorage items and cookies. However, it does not capture HTTP traffic, which is important for web session security analyses; for example, HTTP headers provide useful information about the adoption of defence mechanisms like HSTS. While capturing network traffic could be accomplished just by adding a proxy, a simple proxy would fail to account for Shepherd's awareness of where in the login / logout process it is.

[4] Available at `https://hstspreload.org/`

Our goal is to analyse traffic related to specific phases of session management. Shepherd knows when each phase is reached and thus when traffic should be recorded. We therefore embed a way for Shepherd to enrich the recorded traffic stream with semantic information based on the selenium-wire package.[5] This enables our analysis to exactly target the various phases of session management, and opens the possibility to correlate website interactions (e.g., triggering a button, submitting a form and so on) with their corresponding network traffic.

In this project, we use this functionality in two ways. First, we let Shepherd mark the beginning and the end of each action of the traditional session management process. Second, we introduce marking for interaction steps, such as setting a marker when submitting a form and when the page has stabilised after form submission. This allows re-identification of traffic belonging to an action, which would be lost otherwise. We apply this functionality for traffic reduction. For that, we select actions (e.g., identifying the login page, false login attempts, etc.) that produce irrelevant traffic and remove them from our data set. We tested this in comparison to unfiltered traffic recording and found a reduction of captured traffic in size of up to 65%.

6.3.2 Data collection process

Like in the original Shepherd paper, we extracted the credentials used to access sites from BugMeNot,[6] a website that provides crowd-sourced credentials for other sites. We searched BugMeNot for credentials for 1 million most popular websites according to the Tranco list [LVT+19],[7] which aggregates the ranks from the lists provided by Alexa, Umbrella, Majestic and Quantcast from 14/4/2020 to 13/5/2020. The Tranco list is constructed to provide a more stable list of most popular websites, in contrast to individual rankings [LVT+19]. This resulted in a list of credentials for 56,437 websites.

Data acquisition. We let Shepherd perform the following actions in sequence on these sites:

> connect → identify login area → log in → verify → visit subpages → derive authentication cookies → log out → perform security checks.

In addition to logging out and security tests, we included a step for deep scanning websites. Our goal is to capture authentication cookies that are not immediately set after logging in, or may only be set on subpages [MFK16]. For that, Shepherd extracts URLs from anchor elements that are embedded into the landing page. It first filters third party URLs and duplicates and then picks a random selection of the remaining URLs. Shepherd limits its visits to a maximum of five subpages for performance reasons. We consider a subpage to belong to the same site when its URL shares the `eTLD+1`[8] of the site landing page.

[5] `https://pypi.org/project/selenium-wire/`

[6] `http://bugmenot.com/`

[7] Available at `https://tranco-list.eu/list/VKQN/1000000`

[8] `eTLD+1` includes the eTLD (see `https://developer.mozilla.org/en-US/docs/Glossary/eTLD`) and the next domain part.

Table 6.1: Breakdown of the data collection process

action	# sites	out of	perc.
Connected	53,602	56,437	95%
Login area detected	35,465	53,602	66%
Failed login	28,699	35,465	81%
– *All credentials are invalid*	19,102	28,699	67%
– *CAPTCHA protects login*	2,676	28,699	9%
Logged in	6,766	13,687	49%
– *Authentication cookies identified*	6,124	6,766	91%
Logged out	3,302	6,124	54%

Table 6.1 reports the number of sites reached for the different steps of the data acquisition process, as well as the number of failures for some automatically detected failure cases with large impact. Shepherd's performance in this chapter roughly matches our first study, as presented in Section 4.4, leading to a success rate of 13%. Shepherd found a login area in 35,465 sites (66% of 53,602). Out of those, we found 19,102 sites where all the credentials from BugMeNot turned out to be invalid and 2,676 sites where the login process was protected by a CAPTCHA, hence not amenable for automation. This leaves 13,687 sites where Shepherd had a chance to automate the login process, which succeeded in 6,766 (49%) cases. For most of these cases, we were able to successfully identify their authentication cookies as discussed below. In the following, we restrict our security analysis to the 6,124 sites where login was successful and Shepherd could identify the authentication cookies.

During the data acquisition steps, we captured all requests and responses, with exception of the response body. This resulted in a data set of 86 GB. For each site, we captured cookies, localStorage and sessionStorage in four situations: (1) before logging in, (2) after verifying success of having logged in, (3) after visiting several pages while logged in, and (4) after verifying success of having logged out. In addition, we keep track of which credentials were successfully used to log in, and what URL led to a login area. Once login is verified, we determine which cookies are authentication cookies, that is, cookies without which the browser is no longer logged in. Shepherd's initial implementation relies on the work by Mundada et al. [MFK16] and Calzavara et al. [CTB+14]. The worst case scenario for this approach is an exponential run time with respect to the number of cookies. Therefore, we extended Shepherd to apply the improved solution by Calzavara et al. [CTC+15], which runs in linear time on most sites.

Significance and potential bias. With respect to our discussion concerning the limitations of automated login approaches (cf., Section 4.4.1), any research relying on such a data set should be checked for significance and biases.

To show that our data covers not just random sites from the tail of Tranco, but also very popular sites, we report two interesting results. First, Figure 6.2 shows the distribution of sites for which at least one set of credentials was acquired over the Tranco Top 1M. The detection of invalid credentials is automatically done by Shepherd's "reasonably accurate" integrated detection routines (see Section 4.2.3 and 4.2.4). The figure shows that the most popular sites from Tranco (Top 100K) are

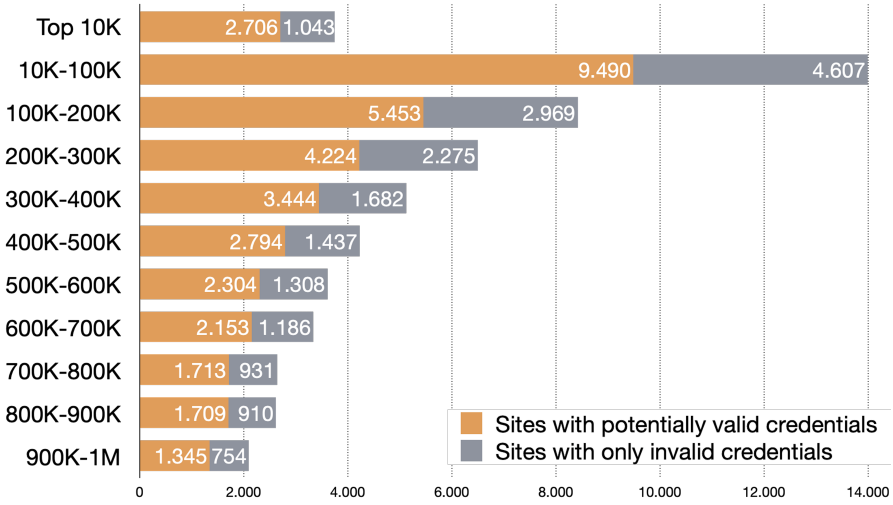

Figure 6.2: Distribution of sites for which at least one set of credentials was acquired over the Tranco Top 1M

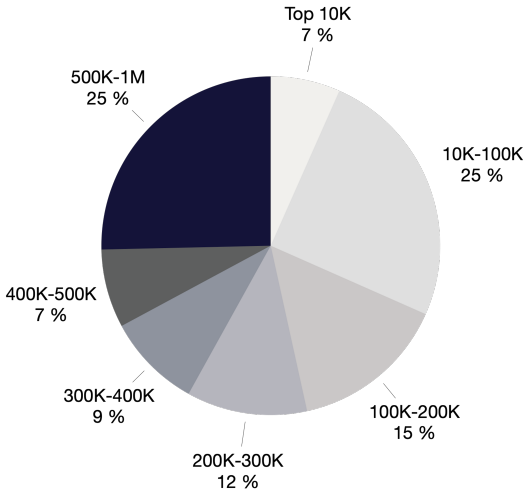

Figure 6.3: Breakdown of successful logins by site popularity

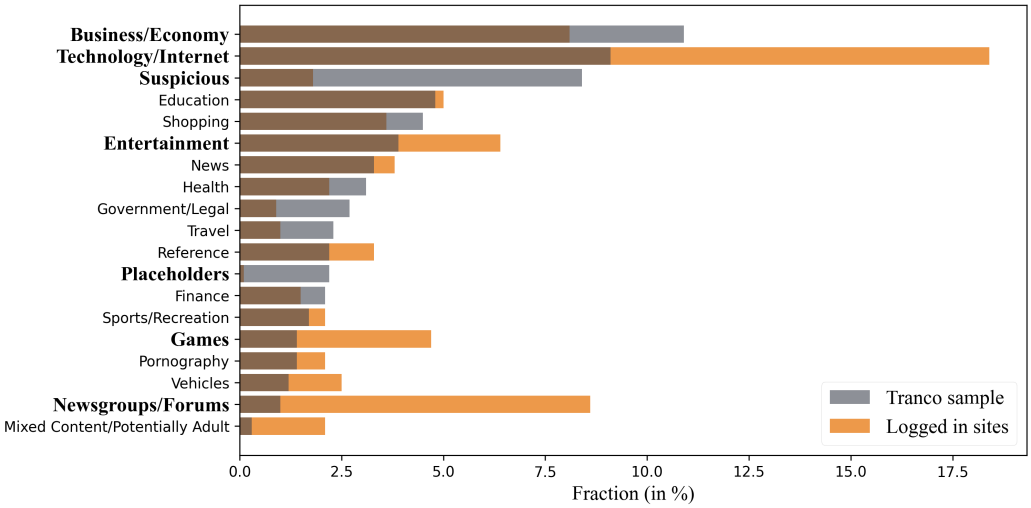

Figure 6.4: Relative frequency for all categories covering more than 2% of either our data set or the Tranco data set. Categories with a difference of over two percentage points between both sets are highlighted in bold

quite represented in BugMeNot. In contrast, Figure 6.3 depicts the distribution of sites with successful logins. This confirms that the data is distributed over the entire Top 1M, with more emphasis on the most popular sites and the first half of the Tranco list.

Next, we investigate the skewness of our data set. Due to the restriction on sites with a public login within our study, we expect an inherent bias. More specifically: not all sites offer a login; such sites are inherently excluded from our study. Moreover, our credential source is crowd-sourced for the goal of avoiding login 'nags' – sites that pester visitors to create a login and limit content available to non-logged in users. We anticipate that this may cause certain types of sites to be underrepresented (e.g., malicious sites), and others, where login nagging is common, to be overrepresented. To gain an estimate of this skewness, we derive categories for sites where Shepherd successfully logged in and compare it with categories of sites in the Tranco list. Specifically, we use Symantec's Review Database [Sym21] which puts sites into 86 categories.[9] Unfortunately, access to Symantec's API is restricted through rate-limits, preventing us from sampling the entire Tranco 1M list. We circumvent this restriction by creating a systematic sample of 50K sites (5% of the Tranco 1M list). We select domains based on a fixed interval (20 ranks), starting from a random position in the top 20 of the Tranco list.

Our results show that our Tranco sample contains sites from all 86 categories, while the login data set covers 79 categories. Notably, missing categories in our data set account for less than 0.1% of all sites. Figure 6.4 depicts the result for categories that exceed a 2% threshold for both data sets. Seven of these categories, marked in bold, differ by more than 2 percentage points between the sets. For these categories,

[9] `https://sitereview.norton.com/#/category-descriptions`

we further discuss why these are over- or underrepresented in our login set:

- **Sites requiring logging in by nature:** Some sites can only be used in their full potential when logging in. Unsurprisingly, we encounter such sites more frequently in our data set. Sites categorised as Games or Newsgroups/Forums are likely candidates that fit this description.

- **Sites usually not shared by users:** Our goal is to investigate the security of legitimate sites targeted at genuine users. In our login data set, two types of sites occur rarely but make up for significant portion in the Tranco list: Suspicious and Placeholders. Since neither category brings value to users, these are less relevant to our study. Moreover, they are also less relevant for genuine users and thus such sites are expected to occur only infrequently in a crowd-sourced data set. We find that this is indeed the case.

- **Tendency in the BugMeNot database:** Sites categorised as Technology/Internet and Entertainment are overrepresented in our login data set (by ∼9 percentage points). We believe this is due to BugMeNot's mission and audience matching these types of sites particularly well.

- **Sites excluded by BugMeNot:** As prescribed in BugMeNot's terms of use,[10] sites that offer paid content may not be submitted. This applies to certain sites in the Business/Economy category.

In conclusion, the prevalence of categories in our data set mostly matches (within ±2%) incidence in the Tranco list. Deviations over this threshold are limited in number and small in size; we thus consider our data set to align sufficiently well with the Tranco set.

In more detail: only 4 out of 86 categories are significantly overrepresented. This is not surprising, as logins are not equally distributed over all categories. Finally, three categories are underrepresented: Business/Economy, Suspicious, and Placeholders. We consider the latter two less relevant for a security study, as neither are meant to provide genuine service to users. In particular, Placeholders sites do not concern real sites, but parked domains, search bait, etc. Similarly, Suspicious sites are sites that seem to be attacking genuine sites or users, not genuine sites themselves. This only leaves the Business/Economy category as underrepresented. The difference for this category is still relatively small (2.8 percentage points). Moreover, despite being underrepresented, it makes up for over 7.5% of our data set. Therefore, there is ample data for this particular category in our data set.

6.4 Login security

The security of web sessions can be broken when the password used for establishing the session is not appropriately protected. We consider two possible attack vectors, which would enable unconstrained impersonation of the victim: *password theft* and *password brute-forcing* enabled by insufficient password strength.

[10] `http://bugmenot.com/terms.php`

6.4.1 Password theft

A number of insecure programming practices might lead to an improper disclosure of passwords over HTTP. In particular, we focus on three prominent attack vectors:

1. If the action of the login form uses the HTTP protocol, the password is communicated in clear, hence even a passive network attacker who just sniffs the network traffic might disclose it. We identified 755 (12%) sites suffering from this vulnerability. Note that we implement this check on the actual login request available in our data set so as to minimize the number of false positives and false negatives, e.g., when the login form is submitted via JavaScript.

2. If the login page is served over HTTP, it can be modified by a network attacker so as to force password leakage, e.g., by changing the action of the login form to HTTP or by injecting an inline script which sends the password to the attacker's website. We identified 901 (15%) sites suffering from this vulnerability.

3. If the password is communicated in the query string of a GET request, it might become part of the URL of the landing page. This means that the password could be leaked as part of the `Referer` header if the landing page loads content over HTTP or from external sites. To spot such cases, we checked the `Referer` header of all the requests made during the website crawl, looking for our password value. We identified 4 sites leaking passwords to third parties (with Google servers being among the third parties in all cases) due to this vulnerability.

Overall, after removing overlaps between classes, we identified 909 (15%) sites exposed to the risk of password theft through the discussed attack vectors. Note that this number is dominated by the second case, i.e., login page served over HTTP. Notwithstanding the significant increase of HTTPS adoption in the last few years, insecurely served login pages remain a key factor of insecurity.

Two points here are worth mentioning about exploitation. First, modern browsers might implement security checks which prevent the introduction or communication of passwords in insecure contexts. However, such checks are not standardised and vary between different browsers, hence we consider bad practices like (1) and (2) as security issues. For example, we observed that while a recent version of Mozilla Firefox (80.0.1) warns users when they fill a login form which is going to be submitted over HTTP, this is not the case for a recent version of Google Chrome (85.0.4183). Moreover, a leakage of secrets via the `Referer` header might be prevented by appropriate configuration of the Referrer Policy header, which provides site operators with the ability of controlling the use of the `Referer` header.[11] However, due to our analysis methodology, we can confirm that all 4 vulnerable sites in the third class leak passwords to external sites via the `Referer` header.

Example: Chip PC. Chip PC Technologies (`www.chippc.com`) is a thin client manufacturer hosting a website to advertise and sell computers. The website provides access to a dashboard where customers can manage orders, warranties and licences. While the website is served over HTTPS, the login form submits authentication credentials to `portal.chippc.com` over HTTP, hence even a passive network attacker

[11] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy`

can sniff passwords just by monitoring the HTTP traffic. This enables impersonation attempts, e.g., the attacker can access the victim's purchase history and steal her product licences.

Example: World Wide Art Resource. World Wide Art Resource (`www.wwar.com`) is a website for artists and creatives who wish to publish their work, with optional paid tiers providing different content hosting plans, exposure and sales commissions. The website uses the GET method to communicate authentication credentials upon login, while importing several libraries from `google-analytics.com`, `consensu.org` and `sharethis.com` domains, in addition to some content from the affiliated website `www.absolutearts.com`. All these different hosts may get access to the passwords of logged in users through the `Referer` header of HTTP requests sent after login.

6.4.2 Password brute-forcing

Even if a password is securely transmitted from the client to the server, it can still be potentially disclosed by a determined attacker if it does not satisfy minimal password strength requirements. The French Data Protection Authority, CNIL, has issued recommendations for securing authentication. CNIL considers four cases, each with their own password requirements:[12] password only, password + account access restrictions, password + additional authentication information, and two factor authentication. Of these cases, our approach can only succeed in logging in for the first two, hence we focus on them and observe that, even in the presence of additional measures such as limiting the number of access attempts, CNIL recommends that the password must contain at least 8 characters from at least 3 of following sets: lowercase letters, uppercase letters, digits and special characters.

Unfortunately, there is no general automated way to detect which password requirements are in place on a given site, since these are not necessarily explicit and can be enforced in different ways. To deal with this problem, we rely on two observations:

1. Although we cannot say anything about general password requirements, we can still check the password strength requirements on the password used to access the web application under analysis, i.e., we can check whether our password is weak or not. This is valuable information for our measurement, since we did not create passwords ourselves, but rather used public passwords from the BugMeNot database, which can be used as a signal of inappropriate password requirements on existing sites.

2. HTML5 provides the `maxlength` attribute to enforce a maximal length for input elements, hence we can inspect its value to assess whether passwords are forced to be shorter than 8 characters. Moreover, HTML5 also supports the `pattern` attribute to enforce that inputs match a given regular expression, which can also be used to infer information about the general shape of accepted passwords.

By combining these two observations, we identified 5,347 (87%) sites using passwords which do not satisfy minimal password strength requirements. The very large majority of our findings comes from the analysis of our own passwords, since the use

[12] `https://www.cnil.fr/sites/default/files/atoms/files/recommandation_passwords_en.pdf`

Table 6.2: Login security results by site popularity

Site popularity	≤1K		≤10K		≤100K		≤1M	
Successful logins	53	100%	430	100%	2,081	100%	6,124	100%
Password theft	0	0%	12	3%	149	7%	909	15%
– login form sent over HTTP	0	0%	8	2%	103	5%	755	12%
– login page served over HTTP	0	0%	10	2%	146	7%	901	15%
– password in query string	0	0%	1	0%	2	0%	4	0%
Password brute-forcing	42	79%	363	84%	1,783	86%	5,347	87%

of the `maxlength` and `pattern` attributes on password fields does not provide much information. In particular, though we identified 884 sites making use of `maxlength` and 25 sites making use of `pattern`, we only found 3 sites where `maxlength` was used to limit a password field to less than 8 characters. The interesting point here is that we are guaranteed that, for those sites, all passwords are weak.

While the use of weak passwords is a bad security practice in general, it does not necessarily constitute an exploitable vulnerability. In particular, websites can implement detection or prevention techniques against brute-forcing attempts, such as locking accounts after a number of failed login attempts. We do not actively test for protection against brute-forcing at scale, as this is ethically dubious at best. In addition, it may violate a site's terms of services and put too much workload on the analysed web applications.

Example: Geeks for Geeks. Geeks for Geeks (`www.geeksforgeeks.org`) is a popular portal offering articles on different technology-related topics, paid courses and hiring help. We have been able to access this site by using a BugMeNot password which is composed just by 4 lowercase letters. This implies that no meaningful password strength requirement is enforced on the website. This is concerning, because the odds of brute-forcing such passwords are realistically high, even if some kind of brute-force mitigation based on the frequency of failed attempts is put in place.

6.4.3 Analysis by popularity

Table 6.2 reports a breakdown of our analysis results by website popularity. The table shows two interesting observations. A positive result is that the most popular websites in our data set do not suffer from the risk of password theft, since no site in the Top 1K leaks passwords in some way. However, the percentage of vulnerable sites monotonically increases when less popular sites are considered, up to a considerable amount (15%). This shows that the most popular sites have a more thorough HTTPS deployment than less popular sites, at least for the purpose of the login process.

Unfortunately, we observe that the use of weak passwords is uniformly widespread and does not significantly correlate with site popularity: the number of vulnerable sites ranges from 79% to 87% in our popularity buckets. This might result from the bias coming from the use of public passwords from the BugMeNot database, since it is plausible that many security-critical sites with strong password requirements are not included there. However, this does not undermine the significance of our finding: there are many popular sites which do not enforce minimal password strength requirements

in the wild. Considering the massive user base of these sites, particularly in the Top 10K bucket, this result is both surprising and concerning.

6.5 Post-login security

Even when users rely on strong passwords which are appropriately protected, session security might be at harm due to the weak security guarantees of cookies in their default configuration. We first consider two traditional attack vectors: *session hijacking*, where the attacker impersonates the victim by stealing their cookies, and *session fixation*, where the attacker impersonates the victim by forcing them to authenticate using a set of attacker-controlled cookies. Finally, we focus on two different types of *cookie brute-forcing* attacks.

6.5.1 Session hijacking via network sniffing

Session hijacking happens when the attacker steals the authentication cookies of the victim and uses them to impersonate them at the target website. Recall that the current design of cookies leaves them susceptible to theft by network attackers, since cookies are normally shared between HTTP and HTTPS, hence potentially exposed in clear over the network. To avoid this, site operators can mark cookies with the `Secure` attribute, which restricts their scope to HTTPS. However, even cookies lacking the `Secure` attribute might be protected against disclosure over HTTP, in particular when the site uses HSTS to enforce the adoption of HTTPS at the client. We find a cookie to have *low confidentiality* against a network attacker when it lacks the `Secure` attribute and either of the following conditions holds true:

1. The server does not activate HSTS. In this case, the attacker can force an HTTP request to the site from the victim's browser and sniff the cookie in clear.
2. The cookie is set for a parent domain and the server activates HSTS without the `includeSubDomains` option. In this case, the attacker can force an HTTP request to a different subdomain of the site to sniff the cookie in clear, as HSTS is only activated for the initial host.

Table 6.3: Confidentiality properties of authentication cookies

	Host-only	Domain
total	1,804	12,087
lacks `Secure` flag	1,300	4,347
– low confidentiality	1,060	4,138

Table 6.3 summarises the confidentiality properties of the authentication cookies collected in our measurement. We observe that 59% and 34% of host-only and domain cookies respectively have low confidentiality against network attackers. Notably, most of the authentication cookies lacking the `Secure` attribute have low confidentiality, which suggests that the current state of the HSTS deployment in the wild is far from satisfying.

We say that a site is vulnerable to session hijacking when *all* its session cookies have low confidentiality, i.e., a network attacker can collect all information required to obtain the authentication cookies and impersonate the victim. In our data set, we identified 1,398 (23%) sites which are subject to this threat. Note that site operators might use defence-in-depth techniques, e.g., browser fingerprinting, to detect stolen session identifiers and terminate hijacked sessions. However, automating this analysis at scale would pose significant technical challenges: for example, sites might keep users authenticated and terminate sessions just when a security-sensitive operation is attempted. We acknowledge this limitation and partially mitigate it by manually confirming successful session hijacking attempts on a random subset of 10 vulnerable sites, including the following.

Example: Sotheby's. The popular auction house Sotheby's runs a website (`www.sothebys.com`) that, while redirecting HTTP requests to HTTPS, does not serve any HSTS header, thus allowing requests to be sent over unencrypted connections. Since none of the site's authentication cookies is marked as `Secure`, a network attacker can just sniff the first HTTP request sent to `www.sothebys.com` and gain access to valid session cookies. Note that the attacker could even force the browser to send such HTTP request by corrupting unrelated HTTP traffic received by the victim's browser.

6.5.2 Protecting against JavaScript cookie stealing

Web attackers may attempt session hijacking by stealing authentication cookies via JavaScript, e.g., exploiting an XSS vulnerability. To mitigate this threat, site operators should apply the `HttpOnly` attribute to their authentication cookies. For the same reasoning as in the previous section, we consider a website as potentially vulnerable against session hijacking via JavaScript cookie stealing when *all* its authentication cookies lack the `HttpOnly` attribute. We find out that out of 6,124 sites in our data set, 2,484 (41%) sites do not set this attribute for any authentication cookie.

Our analysis identifies sites whose authentication cookies lack inherent protection. Note that this lack of protection cannot be turned into an attack without a script injection vulnerability. Nevertheless, it is relevant to analyse cookie protection itself, as XSS is consistently among the most common web security vulnerabilities [OWA17]; furthermore, mitigation techniques like Content Security Policy fail to sufficiently address XSS in practice: up to 94% of policies in the wild do not protect against XSS [WSL+16].

Example: Techrepublic. Techrepublic (`www.techrepublic.com`) is an online news site within the Tranco Top 2K. It uses one cookie for authentication, which is protected against session hijacking attacks via the `Secure` cookie attribute and deployment of HSTS. However, the cookies are not protected against access via JavaScript. This, by itself, does not quite enable session hijacking yet – only scripts in first-party context can access these cookies. Interestingly, Techrepublic includes several third parties in their first-party context, allowing these parties to access user authentication cookies. Finally, the lack of adequate protection of authentication cookies against JavaScript

access means that protection against session hijacking is fully dependent upon a flawless defence against XSS: any XSS flaw in the Techrepublic site can be leveraged to steal authentication cookies.

6.5.3 Session fixation

Session fixation may happen when a website does not refresh the value of the authentication cookies when the privilege level of the session changes, e.g., upon login. In this case, the attacker can force a set of known authentication cookies from the target site into the victim's browser, so as to be able to impersonate the victim after the victim authenticates at the target and gets privileged access to it. To force cookies into the victim's browser, a network attacker can forge HTTP responses from the target site, thus abusing the lack of isolation between HTTP and HTTPS in cookie storage to eventually achieve the same effect as session hijacking.

While refreshing the value of authentication cookies upon login is a best practice, one can also thwart session fixation by ensuring the integrity of session cookies. Specifically, a cookie has *high integrity* against a network attacker when either of the following conditions holds true:

1. The server activates HSTS with the `includeSubDomains` option. In this case, the site forces the use of HTTPS on all the hosts which are allowed to set a cookie for it, thus closing the door to network attacks.
2. The cookie name contains a security prefix (`__Secure-` or `__Host-`), which means the cookie can only be set and accessed over HTTPS.

We say that a site is vulnerable to session fixation when *none* of its session cookies is refreshed upon login and, in addition, *none* of them has high integrity. Interestingly, we found no authentication cookies making use of security prefixes in our data set. This outcome is in line with the observations of a recent study by Calzavara et al. [CFN+19], who found one site using cookie prefixes amongst 10K websites. We identified 1,082 (18%) sites which do not refresh authentication cookies upon login, including 1,011 (16%) sites which are deemed vulnerable to session fixation. The 71 sites which do not refresh authentication cookies, yet still are not vulnerable, all ensure cookie integrity by means of HSTS.

Example: Adult Entertainment Sites. We identified multiple adult entertainment sites vulnerable to session fixation attacks. In most cases, this comes from an inappropriate management of the PHP session cookie `PHPSESSID`. The default PHP session management does not account for logins, as the login logic is site specific. While PHP cannot refresh session identifiers upon login automatically, it offers the `session_regenerate_id` function to be invoked after login to prevent session fixation. It is concerning to find such vulnerabilities in adult entertainment sites, as a successful attack might leak sensitive information.

6.5.4 Cookie brute-forcing

We now focus our attention on two dangerous brute-forcing attacks on cookie values. The first threat we consider comes from the use of predictable identifiers in

session cookies. The risk of brute-forcing attacks may be restricted by rate limiting requests from the same client, or the expiration time of a session, in particular server-side session expiration. Unfortunately, the only way to test whether rate limiting is present, is to exceed the number of allowed requests. We refrain from such an unethical course. Testing server-side session expiration is also non-trivial. As shown in Section 6.6.1, client-side authentication cookies may officially expire long before the server-side session is removed. We are therefore left with considering to what extent cookie value itself is brute-forceable. We use the OWASP recommendations on session ID length,[13] which recommends session identifiers which contain at least 128 bits of entropy. We evaluate this by concatenating all authentication cookies, and computing the entropy of the resulting string. That is, we hold that, in these cases, the attacker can brute-force all the information required to get access to the victim's session. Our crawl identified 1,981 (32%) sites which do not satisfy this security best practice. The average value of entropy among the vulnerable sites is 92 bits, with a standard deviation of 39 bits.

The second threat we consider comes from an infamously insecure practice used for authentication cookie generation: computing the session identifier by applying a potentially invertible function to the password. This allows an attacker who gets access to a session identifier to recompute the password. This is a severe threat as it enables account takeover (via the password change interface) and might lead to impersonation on other services where the password is reused. In particular, we focus on two popular yet now insecure hashing algorithms: MD5 and SHA1. To identify these insecure practices, we compute the MD5 and SHA1 of the password we used to authenticate, and we look for them in the session cookie values. Overall, we identified 63 sites storing a weak hash of the password without salting inside a authentication cookie. Failure to use salting in hashing password results in far greater risk of offline/rainbow tables brute-forcing. We experimentally confirmed that 47 (75%) of these hashes can be trivially inverted into the correct password by using the CrackStation[14] rainbow tables free online service.

Example: DataLife Engine. We found 26 websites storing a weak MD5 hash of the password inside a cookie called `dle_password`, which is the authentication cookie of the DataLife Engine content management system. This is particularly concerning, because all sites built on top of DataLife Engine might improperly disclose passwords. In particular, we identified that in 15 cases the `dle_password` cookie could be sent in clear over HTTP: in 12 cases because the website was served over HTTP, in 3 cases due to the lack of the `Secure` attribute on an HTTPS website without HSTS. All these authentication cookies can be disclosed by network attackers and eventually inverted into the victim's password.

6.5.5 Analysis by popularity

Table 6.4 reports a breakdown of our analysis results by website popularity. The key insight here is that there is no strong correlation between security and popularity. In particular, the percentage of vulnerable sites in the Top 1K bucket is only

[13] `https://owasp.org/www-community/vulnerabilities/Insufficient_Session-ID_Length`
[14] `https://crackstation.net/`

Table 6.4: Cookie security results by site popularity

Site popularity	≤1K		≤10K		≤100K		≤1M	
Successful logins	53	100%	430	100%	2,081	100%	6,124	100%
Session hijacking via network sniffing	9	17%	42	10%	358	17%	1,398	23%
Session hijacking via JavaScript	29	55%	192	45%	888	43%	2,494	41%
Session fixation	6	11%	51	12%	312	15%	1,011	16%
Cookie brute-forcing	13	25%	100	23%	576	28%	2,044	33%
– weak session identifiers in cookies	13	25%	99	23%	564	27%	1,981	32%
– weak password hashes in cookies	0	0%	1	0%	12	1%	63	1%

slightly lower than the percentage of vulnerable sites in the full data set, for all the vulnerabilities we considered.

We find this remarkable, because all the considered vulnerabilities are well-known and have easy solutions, hence we expected site operators at major companies to be aware of these problems and to be able to fix them. In retrospect, however, we see two possible reasons why top sites exhibit more positive figures for login security rather than for cookie security. First, understanding and enforcing login security is easier, since the adoption of HTTPS already fixes the most severe vulnerabilities. Considered how much HTTPS is getting traction, also thanks to the efforts by browser vendors, one might argue that login insecurity has been naturally fixed by the evolution of the web platform over the years. Moreover, based on our research experience, real-world web applications are complex and developed using a number of different technologies. This means that the session management logic is often spread through multiple authentication cookies issued by different components and it might be hard to assess the security of all of them.

6.6 Logout security

Most websites offer users the possibility to terminate sessions by logging out. Though the logout process sounds simple in theory, there are a couple of implementation subtleties which might introduce security flaws. In particular, websites should properly implement both *server-side* and *client-side* session invalidation, as discussed in the following. Server-side session invalidation ensures that terminated sessions are forgotten by the server, i.e., presenting session cookies for those sessions should not enable authenticated access anymore. Client-side session invalidation, instead, guarantees that privacy-sensitive session information is removed from the browser upon session termination.

6.6.1 Server-side session invalidation

The desired effect of a logout is that the session is no longer valid at the server side. If this is not handled properly, an attacker that manages to acquire session identifiers of incorrectly terminated sessions can still get authenticated access to the website. Moreover, unnecessarily extended session validity make session identifiers more vulnerable to the threat of brute-forcing.

In general, checking whether a website has proper server-side session hygiene consists of three steps: (1) login and keep cookies, (2) logout and (3) re-visit the site with the previously stored cookies.

The timing between logging out and revisiting is important. In a properly implemented session management system, server-side session cleanup should (at the latest) coincide with the notification to the client that the session has terminated. However, to account for sites sending a "session terminated" message in parallel with cleaning up session data in their backend servers, we check server-side session invalidation at three different times:

1. Immediately, that is: directly upon page stabilisation[15] after a logout request was sent by the browser. This is how an ideal session management implementation should work.

2. After 5 minutes. This time frame accounts for possible concurrency issues upon session termination, e.g., the logout request needs to be propagated to multiple replicated databases storing session information.

3. After 10 days. This time frame allows us to identify websites where sessions are not invalidated within any reasonable threshold and are definitely at risk.

In the second case, we let Shepherd evaluate every minute if a session is still active. This evaluations stops when the session turns out to be invalid or the five-minute mark is reached. For the final test, Shepherd re-uses the cookie jar from the original login and repeats the login verification step 10 days later.

Overall, we count 2,601 (79%) websites where session cookies were correctly invalidated directly after logout (see Table 6.6). In addition, we found 97 (3%) sites that did not invalidate authentication cookies immediately, yet did so within five minutes. This shows that some tolerance is useful in this kind of analysis. The remaining 604 (18%) sites did not invalidate authentication cookies upon logout within five minutes. Of these, 469 (14%) sites also failed the third test: 10 days later, the session was still valid at the server.

Example: Flattr. Flattr (`www.flattr.com`) is a micro-payment service in the Tranco Top 10K. It enables users to make small (potentially recurring) donations to individuals as a form of patronage. We found that Flattr's authentication cookies were still valid 10 days after logging out. This is unexpected, given the nature of the site (micro-payments). Luckily, Flattr uses several protection measures that prevent cookie stealing, which mitigates the impact of this vulnerability.

Example: Suedkurier. Suedkurier (`suedkurier.de`) is a German regional newspaper with logins (free registration). Using the authentication cookies from the successful login resulted in a logged in state, 10 days after logging out from that session. Moreover, we found that Suedkurier's session identifiers have low entropy. The combination of low entropy and absent server-side invalidation significantly exacerbates the threat of cookie brute-forcing attacks.

[15] A page is considered as stable, when all HTTP responses are fully loaded and the DOM has not been updated for two seconds.

Table 6.5: PII left at client-side after logout.

	cookies_{net}	cookies_{loc}	localStorage
username			
– *regular username*	105	109	14
– *email address*	13	14	16
password	2	2	0
credential*	58	64	17
MD5 username			
– *regular username*	2	2	0
– *email*	2	2	3
MD5 password	0	0	0
MD5 credential*	6	7	0

cookies_{net}: Cookies accessible by network attacker
cookies_{loc}: Cookies accessible by next user attacker
* cases where username = password

6.6.2 Client-side session invalidation

Session invalidation on the client-side serves to avoid data leakage. For example, network attackers can use the attacks from Section 6.5.1 to capture cookies left behind on the client even after session termination. This may leak privacy-sensitive information in case this is contained inside cookies, e.g., an email address. We also consider threats posed by *next user attackers* with access to the same client of the victim, as discussed in Section 6.2.1.

To evaluate proper session clean up, we search for personally identifiable information (PII) in cookies and localStorage items that remain after logging out. In particular, we look for username, email and password in localStorage and in cookie values – both in plain text and hashed with MD5 or SHA1. Note that in our data set, username and passwords sometimes coincide. Thus we cannot always distinguish if the username or password was stored.

Our analysis identified 230 (7%) sites persisting PII in client-side storage after logout. A breakdown of the results according to the different types of client-side storage are shown in Table 6.5. Column $\textbf{cookies}_{net}$ counts cookies which are not protected against network sniffing, hence can be accessed by both types of attackers we consider. Column $\textbf{cookies}_{loc}$ also includes cookies which are locally accessible to the next user attacker alone, while column **localStorage** reports on localStorage items. The table shows that in 186 of the 199 cases (94%), PII is stored in cookies without protection against a network attacker. Similarly worrying, some sites store passwords in cookies, and do not remove these after a logout. We manually verified cases with passwords, and found that insecurity was typically obvious from the cookie name (e.g., `PASSWORD` or `passwd[207860]`). Finally, we also observe that when PII is stored, its value is rarely obscured by means of hashing.

We compare these numbers with PII stored during the login phase. We encountered 756 sites with PII in cookies, which were properly removed upon logout in 557 (74%) cases. We also checked use of localStorage: out of 199 sites storing PII in localStorage, 151 (76%) sites properly cleaned up localStorage upon logout.

Table 6.6: Session invalidation results by site popularity

	≤1K		≤10K		≤100K		≤1M	
logged out	15	100%	169	100%	975	100%	3,302	100%
server-side invalidation:	13	87%	137	81%	819	84%	2,833	86%
– immediately	11	73%	116	69%	734	75%	2,601	79%
– within 5 minutes	1	7%	7	4%	37	4%	97	3%
– 5 minutes – 10 days	1	8%	14	8%	48	6%	135	4%
– unknown, > 10 days	2	13%	32	19%	156	16%	469	14%
client-side left PII behind in:	3	20%	14	8%	78	8%	230	7%
– localStorage	1	7%	6	4%	26	3%	48	2%
– cookies$_{loc}$	2	13%	8	5%	60	6%	199	6%
– cookies$_{net}$	2	13%	8	5%	56	6%	186	6%

Example: Drop APK. Our study revealed a file hoster within the Tranco Top 20K, Drop APK (`dropapk.com`), that keeps track of the username in a user's cookie jar. This cookie is not removed after logging out. For Drop APK, knowledge of the username suffices to list all public files of a user (https://dropapk.to/users/{username}). A next user attacker can exploit this to identify a previous user's username on DropAPK and browse through the public files the user stored on the service.

6.6.3 Analysis by popularity

Table 6.6 reports a breakdown of our analysis results by website popularity. Though the number of sites where we performed our evaluation is relatively small, particularly in the Top 1K bucket, we do not observe any significant correlation between security and popularity. We identified sites incorrectly implementing server-side session termination in all popularity buckets, roughly with the same percentages. Similarly, errors in client-side session invalidation are also fairly constant with respect to popularity (ignoring the limited data for ≤1K).

Interestingly, in all popularity buckets, the next user attacker is only slightly more powerful than the network attacker. This confirms that even top sites often overlook the adoption of cookie protection mechanisms, even for privacy-sensitive cookies. This is concerning, because we expected operators of top sites to be more familiar with the semantics of cookies and their insecure default configuration.

6.7 Perspective

Our approach successfully logged in on 6,124 sites and logged out from 3,302 sites. What we found was quite concerning, at all levels of the session management logic. As to the login phase, we observed insecure connections for sending the login form (15%) or receiving it (12%), passwords leaked to third parties due to being submitted via GET instead of POST (4 sites), widespread (87%) allowance of weak passwords. After login, we identified authentication cookies vulnerable to session hijacking (23%) or accessible via JavaScript (41%), session fixation vulnerabilities (16%), weak session identifiers (32%) and invertible password hashes stored in cookies (47 sites). Finally,

after logout, we found sessions still not invalidated even after 10 days (8%), and failures to purge PII-containing session data from local session storage (8%).

Despite the bias coming from the analysis of sites for which valid access credentials can be found in a public database like BugMeNot, our results paint a troubling picture of the current state of the Web, because most of sites we analysed are unquestionably popular services ranking in the Tranco Top 100K [LVT+19]. Although all the vulnerabilities we identified are relatively well known to web security experts, they are not necessarily easy to deal with and we recommend actions at many different layers to improve on the current state of affairs.

The first observation we make is that the login process is arguably the easiest part to secure of the session management logic. Security-savvy web users can largely mitigate the dangers coming from insecure login pages. In particular, users can leverage password managers to generate strong passwords even for sites which accept weak passwords, and they can install popular browser extensions like HTTPS Everywhere[16] to force the adoption of HTTPS even on sites which do not deploy HSTS. We observe that browser vendors can play a major role to improve login security and they are already taking actions in this direction. For example, the most recent versions of Google Chrome warn users when passwords are communicated in clear over HTTP and most modern browsers already ship an integrated password manager. We think and hope that by further pushing these actions it will be possible to rule out insecure logins from the Web within a reasonable time frame.

Unfortunately, despite their apparent simplicity, web session security issues occurring after login are much harder to fix. There are several reasons for this. First, cookies are opaque to both web users and browser vendors, so detecting authentication cookies to analyse (and automatically improve) their security guarantees requires custom heuristics [CTC+15]. In particular, the most effective heuristics operate online (via testing) and are not straightforward to implement in commercial browsers without sacrificing performance or compatibility with existing web applications. In principle, one could try to experiment with safe defaults, e.g., automatically promote all cookies to `Secure`, however such forms of client-side protection can break existing websites [BCF+15]. In the end, we believe that secure session management crucially relies on the intervention of site operators, i.e., browser vendors and web users are limited in their range of actions. Automated security scanners like our extension of Shepherd are thus an important tool to improve the current state of web session security.

6.8 Trends and comparison with related work

Web session security is a wide research area, whose key contributions were summarised in a relatively recent survey [CFS+17]. Here, we discuss selected prior work which is most closely related to ours, and we describe trends based on previously conducted session security evaluations.

[16] `https://www.eff.org/https-everywhere`

Table 6.7: Comparison of post-login studies investigating aspects of session security

	[MFK16]	[DIP20]	this work
logging in stats			
– login	manual	automated	automated
– # of sites approached	149	1.6M	53.6K
– # of successful logins	149	25.2K	6.1K
login security			
– password theft	–	–	✓
– password brute-forcing	–	–	✓
post-login security			
– session hijacking via network sniffing	✓	✓	✓
– session hijacking via JavaScript	✓	✓	✓
– session fixation	–	–	✓
– cookie brute-forcing	✓	–	✓
logout security			
– server-sided session invalidation	✓	–	✓
– session data clean-up	✓	–	✓
privacy			
– personal data leakage	–	✓	–

6.8.1 Comparison with closely related work

Only two previous studies assess web session security after logging in with an semi-automated or automated approach: a first study by Mundada et al [MFK16], and a second study by Drakonakis et al. [DIP20]. Table 6.7 compares the aspects investigated by these studies and ours. The study by Mundada et al. [MFK16] uses a manual login approach; users carry out the login process, while the security assessment is automated. Due to the manual login process, their corpus is much smaller than either Drakonakis et al.'s work, or ours: only 149 sites have been analysed. Drakonakis et al.'s study relies on account creation and logging in with SSO. This approach to automatically logging in has a low success rate. They compensate for the low success rate by attempting logins on the largest number of sites of all three studies, i.e., around 1.6M sites.

With respect to security analyses, there are several noteworthy differences between these studies. The overlap between the security assessment of Drakonakis et al. and our work concerns session hijacking via network sniffing and protection against JavaScript cookie stealing. Though there is some overlap between Mundada et al.'s work and our security analysis in terms of threats, there are significant differences with our work. Their work primarily focuses upon automated detection of session cookies, rather than measuring web session security at scale (they only focus on 149 sites, due to limited login automation). As such, they do not evaluate login security and session fixation.

Other studies focused on specific web session security problems. For example, session hijacking has been studied against different threat models, including web attackers [NMY+11], network attackers [SPK16b] and both [BCF+15]. Session fixation also got some attention by the research community, particularly with the design of possible defence mechanisms [JBS+11; dRND+12]. In more recent work, Calzavara et al. proposed black-box testing strategies to identify security flaws in web sessions, including session hijacking and session fixation [CRB19]. However, the experimental

analyses in all these papers are either small-scale (in the order of tens of sites) or based on data collected without logging in, which limits the analysis surface and requires one to come up with unreliable heuristics for authentication cookie detection [CTC+15].

The only research study on login security on the Web is due to Van Acker et al. [vAHS17]. They also discuss bad practices which enable exploitation by network attackers, e.g., login pages served over HTTP or sending the password in clear. However, their analysis methodology is different from ours, since they collect login forms by inspecting the HTML rather than by dynamically monitoring form submissions, which is generally more precise. For example, dynamic monitoring naturally covers the case of form submission via JavaScript, which was not handled in [vAHS17].

Compared to the security of login pages, more attention was given to the creation of passwords which are resilient to brute-forcing attacks [HA12; SBC+15; SKD+16; SW19].

In our work, we base our analysis on standard recommendations from CNIL, which appear to be widespread based on anecdotal evidence. For example, the popular LastPass[17] password manager generates passwords which follow the CNIL password strength requirements in its default configuration.

6.8.2 Trends in adoption of security measures

In the last decade, several studies have presented data on the current state of selected aspects of session security. Approach, measurements taken, and interpretation all vary significantly between these studies. Nevertheless, there is some overlap in the underlying security measures they sampled. This makes it possible to determine adoption trends in the last decade. Table 6.8 lists findings from studies on session security aspects until 2022. The general finding is that adoption rates for these simple server-side security measures are slowly increasing, though still far from ubiquitous. This section continues with a discussion of trends for specific measures and the findings of our study in Chapter 6.

Adoption of the HttpOnly cookie attribute. Data on the adoption of the `HttpOnly` cookie attribute has been reported in [SPK16b; MFK16; DIP20; ZE10; NMY+11; CUT+21; RCW+22]. Since the reported numbers vary with each study's data set, these should be considered as a rough indicator for adoption. Reports before 2016 point to a low adoption rate between 22% and 63% at most. In comparison, later studies indicate an increase to a rate between 56% and 77% in a best-case scenario.

Adoption of the Secure cookie attribute. Multiple reports [BCF+15; BSB+12; MFK16; DIP20; CUT+21; RCW+22] provide data on the adoption of the `Secure` cookie attribute. As these reports differ in how they report results (e.g., for partial or all cookies for the entire site), they are not directly comparable. Nevertheless, the overall trend is clearly upwards, from a low of 45% in 2015 [BCF+15] to a vastly improved – but still disconcertingly low – 59% in our work. Do note that we find susceptibility to session hijacking significantly lower at 23%. This is due to websites that deploy the `Secure` cookie attribute to some (but not all) authentication cookies and due to the deployment of other security measures, such as HSTS.

[17] `https://www.lastpass.com`

Table 6.8: Trends in adoption of security measures (in % of sites)

year	#sites	logs in	cookie attributes			invalidation	
			HttpOnly	Secure	HSTS	server	client
2010, [ZE10]	50	✓	48%	–	–	–	–
2011, [NMY+11]	419K	–	22%	–	–	–	–
2012, [BSB+12]	64	✓	–	48%	–	69%	–
2015, [KB15]	1.1M	–	–	–	1%	–	–
2015, [BCF+15][1]							
– 2014, from [CTB+14]	70	✓	63%	8%	–	–	–
– 2015 #1	1K	–	28%	5%	–	–	–
– 2015 #2	≤100	✓	38%	20%	–	–	–
2016, [SPK16b]	26	✓	62%	–	–	–	47%
2016, [MFK16]	149	✓	68%	57%	[2]57%	50%	91%
2016, [SKP16]	22K	–	–	–	11%	–	–
2019, [DIP20]	25K	✓	77%	57%	[2]58%	–	–
2020, [CoSe21] (our work)	**6K**	✓	**59%**	[1]**59%**	[1,2]**63%**	**79%**	**93%**
2021, [CUT+21]	15K	–	56%	45%	[2]43%	–	–
2022, [RCW+22]	10K	–	31%	29%	[2]45%	–	–

[1]: Numbers are reported in cookies and not sites
[2]: Numbers apply only to sites without protection of the `Secure` cookie attribute

Adoption of HTTP Strict Transport Security. Previous reports on the adoption of HSTS show an overall small adoption: 1% of sites found by Kranch and Bonneau [KB15] in 2015 and 11% by Sivakorn et al. [SKP16] in 2016. Thankfully, adoption rates have picked up in recent years, culminating in a 63% adoption rate in our study. Nevertheless, all studies that investigated HSTS consistently found that lack of a `Secure` cookie attribute is only rarely mitigated by HSTS.

Server-side session invalidation. To the best of our knowledge, there are only two prior studies providing data on session invalidation, the study by Bursztein et al. [BSB+12] and the study by Mundada et al. [MFK16]. Both studies have limited sample size: 64 sites and 149 sites, respectively. As such, we cannot extrapolate from these studies, but we do note that a lack of server-side invalidation frequently occurred in either study. Our results suggests that the trend is improving, though we still find every fifth site failing to properly invalidate authentication cookies on the server-side following logout.

Session data clean up after logging out. Sivakorn et al. [SPK16b] conducted an in-depth study for privacy leakage on a small number of websites. As such, they evaluated if various privacy leaks even occur after logging out. They found that 47% of the assessed sites delete cookies holding PII. Mundada et al. [MFK16] also performed tests for client-side cleanups. In contrast, they looked for authentication cookies that remain on the client-side after logging out. In their study, 91% of sites removed such cookies. Our study shows that this issue has further decreased.

6.9 Conclusions

We set out to investigate the current state of web session security in the wild, by performing a comprehensive session security analysis based on post-login data collected at a large scale. We used the Shepherd framework for post-login studies to automate logins, and extended it to handle logouts and capture traffic for further analysis. We acquired the needed credentials from a crowd-sourced repository (BugMeNot). We analysed security of the login process, security of the session (and its cookies), and security of the logout process. This includes an analysis of password strength of accepted passwords in practice, and the first (to the best of our knowledge) large-scale analysis of session invalidation.

As future work, we plan to further improve the automation of the logout process based on the data collected in the present study. We also want to further extend the scale of our analysis by integrating support for SSO, which would allow us to collect information from sites which are not included in BugMeNot. Finally, we would like to investigate how to extend our security analysis to other attackers, such as web attackers, without biasing the results towards overly conservative assumptions, e.g., that all web applications might suffer from XSS or other script injections.

Part II

Advancing Attacks and Defences

Chapter 7

Specific Detection of Web Bots

Web bots are used to automate client interactions with websites, which facilitates large-scale web measurements. At the same time, they allow to automate attacks, which poses a threat to online services. As a defence, websites may employ web bot detection. When they do, their response to a bot may differ from responses to regular browsers. The discrimination can result in deviating content, restriction of resources or even the exclusion of a bot from a website. This places strict restrictions upon studies: the more bot detection takes place, the more results must be manually verified to confirm the bot's findings.

To investigate the extent to which bot detection occurs, we reverse-analysed commercial bot detection. We found that in part, bot detection relies on the values of browser properties and the presence of certain objects in the browser's DOM model. This part strongly resembles browser fingerprinting. We leveraged this for a generic approach to detect web bot detection: we identify what part of the browser fingerprint of a web bot uniquely identifies it as a web bot by contrasting its fingerprint with those of regular browsers. This leads to the fingerprint surface *of a web bot. Any website accessing the fingerprint surface is then accessing a part unique to bots, and thus engaging in bot detection.*

We provide a characterisation of the fingerprint surface of 14 web bots. We show that the vast majority of these frameworks are uniquely identifiable through well-known fingerprinting techniques. We design a scanner to detect web bot detection based on the reverse analysis, augmented with the found fingerprint surfaces. In a scan of the Alexa Top 1 Million, we find that 12.8% of websites show indications of web bot detection.

This chapter is based on the following publication:

Fingerprint Surface-Based Detection of Web Bot Detectors. Hugo Jonker, Benjamin Krumnow, and Gabry Vlot. *In Proc. 24th European Symposium on Research in Computer Security (ESORICS'19)*, DOI: 10.1007/978-3-030-29962-0_28, 2019, [ESORICS19].

7.1 Introduction

Web bots may be used for benign purposes, such as search engine indexing or research into the prevalence of malware. They may also be used for more nefarious purposes, such as comment spam, stealing content, or ad fraud. Benign websites may wish to protect themselves from such nefarious dealings, while malicious websites (e.g., search engine spammers) may want to avoid detection. To that end, both will deploy a variety of measures to deter web bots.

There is a wide variety of measures to counter bots, from simple countermeasures such as rate limiting to complex, such as behavioural detection (mouse movements, typing rates, etc.). The more different a web bot is, the simpler the measures needed to detect it. However, modern web bots such as Selenium allow a bot to automate the use of a regular browser. Such a web bot thus more closely resembles a regular browser. To determine whether the visitor is a web bot may still be possible, but requires more information about the client side. Interestingly, more advanced countermeasures allow a website to respond more subtly. Where rate limiting will typically block a visitor, a more advanced countermeasure may (for example) omit certain elements from the returned page.

A downside of detection routines is that they affect benign web bots as much as malicious web bots. Thus, it is not clear whether a web bot 'sees' the same website as a normal user would. In fact, it is known that automated browsing may result in differences from regular browsing (e.g., [WD05; WSV11; ITK+16]). Currently, the extent of this effect is not known. Nevertheless, most studies employing web bots assume that their results reflect what a regular browser would encounter. Thus, the validity of such studies is suspect.

In this work, we investigate the extent to which such studies may be affected. A website can only tailor its pages to a web bot, if it detects that the visitor is indeed a web bot. Therefore, studies should treat websites employing web bot detection differently from sites without bot detection. This raises the question of how to detect web bots. We have not encountered any studies focusing exclusively on detecting whether a site uses web bot detection.

Contributions. In this chapter, we devise a generic approach to detecting web bot detection, which leads to the following four main contributions.

- We conduct a reverse analysis of a commercial client-side web bot detector. From this, we observe that specific elements of a web bot's browser fingerprint are sufficient to lead to a positive conclusion about a script. This, in turn, suggests that the browser fingerprint of web bots is distinguishable from the browser fingerprint of regular browsers – and that (some of) these differences are used to detect web bots.

- We create a setup to capture all common differences of web bots detectable via browser fingerprint. We call this collection of fingerprint elements that distinguish a web bot from a regular browser the *fingerprint surface*, analogous to Torres et al. [TJM15]. We use our setup to determine the fingerprint surface of 14 popular web bots.

- We design a bot-detection scanner and scan the Alexa Top 1 million for fingerprint-based web bot detection. To the best of our knowledge, we are the first to assess the prevalence of bot detection in the wild.
- Finally, we provide a qualitative investigation of whether websites tailor content to web bots.

Availability. The results of the web bot fingerprint surfaces and the source code used for our measuring the prevalence of web bot detectors are publicly available for download from [ART-ESORICS19].

Outline. In the following, we conduct a reverse analysis of a commercial bot detector (Section 7.2). We use our insights from this analysis to develop a methodology for building a bot's fingerprint surface (Section 7.3) and apply it to popular automation frameworks (Section 7.4). Based on the identified properties that appear only in automation frameworks, we visit one million websites to detect client-side bot detection (Section 7.5). Then, we test the effect of bot detection against an automated headless and native browser (Section 7.6). Finally, we conclude our investigation of fingerprint-based bot detection (Section 7.7).

7.2 Reverse analysis of a commercial web bot detector

In our search for sites that engage in web bot detection, we encountered a site that allegedly can detect and block Selenium-based visitors [Wei15]. We verified that this site indeed blocks Selenium-based visitors by visiting the site with user- and Selenium-ChromeDriver-driven browsers systematically. We investigated JavaScript files used on this site and analysed the page's traffic. The traffic analysis showed that several communications back to the host contained references to 'distil', e.g., in file names (`distil_r_captcha_util.js`) or in headers (`X-Distil-Ajax: ...`). This was due to two of the scripts originating from Distil Networks, a company specialised in web bot detection, and thus the likely cause of the observed behaviour. We manually de-obfuscated these scripts by using a code beautifier and translating hex-encoded strings, after which we could follow paths through the code. This allowed us to identify a script that provided the following three main functionalities:

Behaviour-based web bot detection. We found multiple event handlers added to JavaScript interaction events. These cover mobile and desktop-specific actions, such as clicks, mouse movements, a device's orientation, motion, keyboard and touch events.

Code injection routines. The traffic analysis revealed frequent communication with the first party server. Within this traffic we found fingerprint information and results of web bot detection. This would allow a server to carry out additional server-side bot detection. We further identified routines that enable the server to inject code

```
// Example of original, obfuscated code:
// array containing literals used throughout the source code
var _ac = ["\x72\x75\x6e\x46\x6f\x6e\x74\x73",
  "\x70\x69\x78\x65\x6c\x44\x65\x70\x74\x68", .....]

// Example of de-obfuscation
// obfuscated: window[_ac[327]][_ac[635]][_ac[35]](_ac[436])
// de-obfuscated:
window[document]["documentElement"]["getAttribute"]("selenium")

// Example of de-obfuscated and beautified code
sed: function() {
  var t;
  t = window["$cdc_asdjflasutopfhvcZLmcfl_"] || \
  document["$cdc_asdjflasutopfhvcZLmcfl_"] ? "1" : "0";

  var e;
  e = null != window["document"]["documentElement"]\
   ["getAttribute"]("webdriver") ? "1" : "0";
  ...
}
```

Listing 7.1: Examples from bot-detection script.

in response to a positive identification. In our test, this resulted in a CAPTCHA being included on the page.

DOM properties-based web bot detection. Lastly, we found that multiple built-in objects and functions are accessed via JavaScript (e.g., see Listing 7.1). Some of the properties accessed thusly are commonly used by fingerprinters [TJM15]. We also found code to determine the existence of specific bot-only properties, such as the property `document.$cdc_asdjflasutopfhvcZLmcfl_` (a property specific to the ChromeDriver). Keys from the `window` and `document` objects were acquired by Distil. Moreover, a list of all supported mime types was also collected (via `navigator.MimeTypes`).

Moreover, we investigated whether changing the name of this specific property affects bot detection. We modified ChromeDriver to change this property's name and used the modified driver to access the site in question 30 times. With the regular ChromeDriver, we always received "bot detected" warnings from the second visit onwards. With the modified ChromeDriver, we remained undetected.

7.3 A generic approach to detecting web bot detection

From the reverse analysis, we learned that part of Distil's bot detection is based on checking the visitor's browser for properties. Some of these properties are commonly used in fingerprinting, others are unique to bots. Moreover, in testing with a modified ChromeDriver, we found that the detection routines were successfully deceived by only changing one property. This implied that at least some detection routines used by Distil fully rely on specifics of the browser fingerprint. Moreover, both FPDetective [AJN+13] and OpenWPM [EN16] checked whether a website accesses specific

browser properties. By combining these findings, we develop an approach to detecting Distil-alike bot-detection on websites.

To turn this into a more generic approach that will also detect unknown scripts, we expand what properties we will scan for. The properties that are used to detect a web bot will vary from one web bot to another. To detect web bot detection for a specific web bot, we first determine its fingerprint surface and then incorporate those properties that are in its fingerprint surface into a dedicated scanner. Remark that properties and variables that are unique to the fingerprint of a specific web bot serve no purpose on a website, unless the website is visited by that specific web bot and the site aims to change its behaviour when that occurs. Therefore, we hold that if a portion of a fingerprint is unique to a web bot, any site that checks for or operates on that portion is trying to detect that web bot.

With that in mind, we designed and developed a scanner based on the discovered fingerprint surfaces. This scanner thus allows us to scan an unknown site and determine if it is using fingerprint-based web bot detection.

Note that this design does not incorporate stealth features to hide its (web bot) nature from visited sites. To the best of our knowledge, this is the first study to investigate the scale of client-side web bot detection. As such, we expect web bot detection to focus on other effects than hiding its presence. Therefore, we nevertheless deemed this approach sufficient for a first approximation on the scope of client-side web bot detection.

7.4 Fingerprint surface of web bots

A fingerprint surface is that part of a browser fingerprint that distinguishes a specific browser or web bot from any other. A naive approach to determining a fingerprint surface is then to test a gathered browser fingerprint against all other fingerprints. However, layout engine and JavaScript engine tend to be reused by browsers. The fingerprints of browsers that use the same engines will have large overlap. Thus, to determine the fingerprint surface, it suffices to only explore the differences compared to browsers with the same engines. For example: Chrome and Chromium contain the property `document.$cdc_asdjflasutopfhvcZLmcfl_` only when used via ChromeDriver, otherwise not.

Thus, we classify browsers and web bots into browser families, according to the used engines. We then gathered the fingerprint of a bot-driven browser, and compared it with regular browsers from the same family. Only properties that are unique to the web bot in this comparison are part of its fingerprint surface. Interestingly, we found that every browser that uses the same rendering engine, also uses the same JavaScript engine. Thus, for the examined browsers, no fingerprint differences can arise from differences in JavaScript engine.

We set up a website to collect the various fingerprints. To that end, we extended fingerprintJS2,[1] a well-known open source browser fingerprinting package, as discussed below. We visited this site with a wide variety of user- and bot-driven browsers to determine the fingerprint surfaces of 14 web bots.

[1] `https://github.com/Valve/fingerprintjs2`

browser engine	regular browser	automated browser
Webkit	Safari Chrome (v1–v26)	PhantomJS, Selenium + WebDriver
Blink	Chrome (v27+) Chromium Opera (v15+) Edge (from mid-2019)	Puppeteer + Chrome NightmareJS Selenium IDE Selenium + WebDriver
Gecko	Firefox	Selenium IDE, Selenium + WebDriver
Trident	Internet Explorer	Selenium + WebDriver
EdgeHTML	Edge (till mid-2019)	Selenium + WebDriver

Table 7.1: Classification of browsers based upon browser engine

7.4.1 Determining the browser engine of web bots

In our classification, we omitted bot frameworks that do not use a complete rendering engine to build the DOM tree (see HTTP engine interfaces in Section 2.2). We included frameworks popular amongst developers and/or in literature at the time of writing, specifically:

- PhantomJS: a custom (headless) browser based on WebKit, the layout engine of Safari. PhantomJS is included as it is used in multiple academic studies, even though its development has been suspended.[2]
- NightmareJS: a high-level browser automation library using Electron[3] as a browser. It allows to be run in headless mode or with a graphical interface.
- Selenium and WebDriver: a remote control for web browsers. There are specific drivers for each of the major browsers.
- Selenium IDE: Selenium available as plugin for Firefox and Chrome.
- Puppeteer: a NodeJS library to control Chrome and Chromium browsers via CDP. CDP allows to instrument Blink-based browsers.

This leads to the classification shown in Table 7.1. Browsers based on different browser engines use different rendering and JavaScript engines, which will lead to differences in their browser fingerprints. However, all browsers within one browser engine use the same rendering and JavaScript engines. This means their browser fingerprints are comparable: differences in these fingerprints can only originate from the browsers themselves, not from the underlying engines.

[2] `https://github.com/ariya/phantomjs/issues/15344`

[3] Electron is a framework for making stand-alone apps using web technologies. It relies on Chromium and Node.js.

7.4.2 Determining the fingerprint surface

We use the above classification of browser families to determine the fingerprint surface of the listed web bots. To determine the complete fingerprint surface is infeasible, as already noted by Nikiforakis et al. [NKJ+13] and Torres et al. [TJM15]. To wit: a fingerprint is a form of side channel for re-identification. As it is infeasible to account for all unknown side channels, it is not feasible to establish a complete fingerprint surface. Hence we follow a pragmatic approach to identifying the fingerprint surface (much like the aforementioned studies). We use an existing fingerprint library and extend[4] it to account for the additional fingerprint-alike capabilities encountered in the analysis of the commercial bot detector, listed below, as well as best practices for bot detection encountered online. The fingerprint surface collected by the tool is shown in Table 7.2. The updates added due to the reverse analysis are:

- All keys from the `window` and `document` objects.
- A list of all mimetypes supported by the browser.
- A list of all plugins supported by the browser.
- All keys and values of the `navigator` object.

In addition, there are several discussions on best practices for identifying web bots available online. From this, we included the following extra elements to include in the browser fingerprint:

- Lack of "bind" JavaScript engine feature [She15b]:
  Certain older web bots make use of outdated JavaScript engines that do not support this feature, which allows them to be distinguished from full JavaScript engines.
- Stack trace [She15a]:
  When throwing an error in PhantomJS, the resulting stack trace includes the string 'phantomjs'.
- Properties of missing images [Vas17]:
  The width and height of a missing image is zero in headless Chrome, while being non-zero in full Chrome.
- Sandboxed `XMLHttpRequest` [She15b]:
  PhantomJS allows turning off "web-security", which permits a website to execute a cross-domain `XMLHttpRequest()`.
- Autoclosing dialog windows [She15b]:
  PhantomJS auto-closes dialog windows.

The test site hosting this fingerprint script was then visited with each browser and web bot with the same browser engine. Only properties that differed between browsers and bots with the same browser engine constitute elements of those bots'

[4] `https://github.com/bkrumnow/BrowserBasedBotFP/blob/master/public/js/fingerprint.js`

– Plugin Enumeration	– CPU	– DNT User Choice
– Font Detection	– OpenDatabase	– Flash Enabled
– User-Agent	– Canvas fingerprint	– colorDepthKey
– HTTP Header	– Mime-type Enumeration	– HTML body behaviour
– ActiveX + CLSIDs	– IndexedDB	– Physical px ratio to CSS px
– Device Memory (RAM)	– WebGL Fingerprinting	– **Window object keys**
– Screen resolution	– AdBlock	– **Document object keys**
– Available screen resolution	– Language inconsistencies	– **Navigator properties**
– Browser Language	– OS inconsistencies	– ***StackTrace***
– DOM Storage	– Browser inconsistencies	– ***Missing image properties***
– IP address	– Resolution inconsistencies	– ***Sandboxed `XMLHttpRequest`***
– Navigator platform	– Timezone	– ***Autoclosing dialogs***
– Touch support	– Number of cores	– ***Availability of bind engine***

Table 7.2: Browser fingerprint gathered. Newly added properties are marked in bold. Bold italic elements resulted from discussions on best practices.

fingerprint surfaces. The versions and setup that were used during our experiment are listed in Table 7.3. Human-controlled browsers are marked as bold.

Remark that not all deviations in fingerprint lead to a fingerprintable surface. For example, an automated browser may offer a different resolution from a regular browser, which is nevertheless a standard resolution (e.g., 640x480). We thus manually evaluated the resulting deviations between the fingerprints of one browser family and, for each web bot, determined its fingerprint surface accordingly.

OS	**browser**	**version**
Ubuntu 16.04	**Chrome**	64.0.3282.140
	Chrome & Chromium + Selenium ChromeDriver	Sel: 4.0.0; WD: 2.35
	Chrome + Selenium IDE	C: 62.0.3202.94; IDE: 3.0.1
	NightmareJS	2.10
	Chrome & Chromium Puppeteer	1.1.0
	Firefox	54.0.1
	Firefox + Selenium GeckoDriver	Sel: 4.0.0; WD: 0.19.1
	Firefox + Selenium IDE	2.9.1 & 3.0.1
	Opera	53
	Opera + Selenium OperaDriver	Sel: 4.0.0; WD: 2.36
Windows 10	**Microsoft Edge**	41.16.299.15.0
	Microsoft Edge + Selenium Edge Driver	Sel: 4.0.1; WD: 10.0.16299.15
	Internet Explorer	11.0.50
	IE + Selenium InternetExplorerWebDriver	Sel: 4.0.1; WD: 3.9.0
OS X 10.13.5	**Safari**	12.0
	SafariDriver	Sel: 3.6.0; WD: N/A
	PhantomJS	2.1.0

Table 7.3: Configurations used to determine fingerprint surfaces.

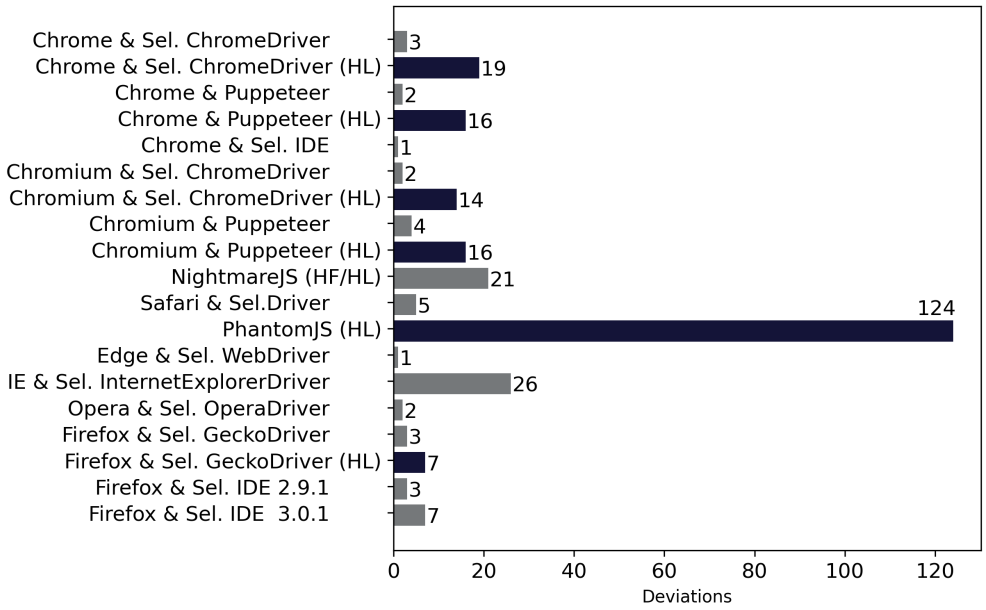

Figure 7.1: Browser-wise comparison of the number of deviations. Bars for headless browsers are depicted in black

7.4.3 Resulting fingerprint surfaces

Several web bots support *headless* (HL) mode. This mode functions similarly to normal operation of the web bot, but does not output its results to a screen. In total, we determined the fingerprint surfaces of 14 web bots (full fingerprint surfaces available online[5]). Together with variants due to HL mode, this resulted in 19 fingerprint surfaces. We found both newly introduced properties and existing properties where the bot-browser has distinctive values. Figure 7.1 depicts the number of deviations (i.e., the number of features in the identified fingerprint surface) of the tested web bots. As can be seen, PhantomJS has many deviations. Another finding is that headless mode leads to a greater number of deviations. This happens for all web bots except for NighmareJS.

The results of our fingerprint gathering differed on several points from the results of the reverse analysis. Specifically: for several of the tests used by Distil, we did not encounter any related fingerprint. We investigated this discrepancy by conducting source code reviews of web bot frameworks to trace such tests back to specific web bot frameworks. We found several properties,[6] that were no longer in the versions of the frameworks we tested, but were present in other versions (older versions or derived versions such as Selendroid). This underscores the incompleteness of the derived fingerprint surface: updates to web bot frameworks will thus result in changes to the fingerprint surface.

[5] `https://bkrumnow.github.io/fpbotdetection/fingerprints.html`

[6] *webdriver_evaluate webdriver-evaluate*, *fxdriver_unwrapped*, *$wdc*, *domAutomation* and *domAutomationController*

Table 7.4 shows an example of a set of deviations and the resulting fingerprint. It lists deviations found by comparing Chrome with a headless Selenium-Webdriver-driven Chrome browser. The deviations listed under the UserAgent string (equally to request headers), window and document keys are unique properties and values, that together build the fingerprint surface. Other properties, such as missing plugins or screen resolutions, might be useful indicators for a detector, but are not unique for web bots.

7.5 Looking for web bot detectors in the wild

In this section, we use the identified web bot fingerprint surfaces to develop a scanner that can detect web bot detectors. Since the fingerprint surfaces are limited to the web bots we tested, we extended our set of fingerprint surfaces with results from the reverse analysis and other common best fingerprinting-alike practices to detect web bots. The resulting fingerprint features were expressed as patterns, which were loaded into the scanner. The scanner is built on top of the OpenWPM web measurement framework [EN16]. OpenWPM facilitates the use of a full-fledged browser controllable via Selenium. The scanner thus resembles a regular browser and cannot be distinguished as a web bot easily without client-side detection. Moreover, OpenWPM implements several means to provide stability and recovery routines for large-scale web measurement studies.

We set up the scanning process as follows: first, the scanner connects to a website's main page and retrieves all scripts that are included by *src* attributes. Each script that matches at least one pattern is stored in its original form, together with the matched patterns and website metadata. Scripts that do not trigger a match are discarded.

7.5.1 Design decisions

Some parts of the fingerprint surface concern not properties, but their values. For example, in Table 7.4, the value of `navigator.useragent` contains 'HeadlessChrome' for a web bot, instead of 'Chrome' for the regular browser. To detect whether client-side scripting checks for such values, we use static analysis. To perform static analysis, the detection must account for different character encodings, source code obfuscation and minified code. Therefore, the scanner transforms scripts to a supported encoding, removes comments and de-obfuscate hexadecimals. The resulting source code can be scanned for patterns pertaining to a specific web bot's fingerprint surface.

Note that our approach has several limitations. In the current setup, the scanner does not traverse websites. As such, data collection is limited to scripts included on the first page. We caution here once again that browser fingerprinting is only one of a handful of approaches to detecting bots. For example, this approach cannot detect behavioural detection. Nevertheless, from the reverse analysis we learned that browser fingerprinting by itself can be sufficient for a detector script to conclude that the visitor is a web bot, irrespective of the outcome of other detection methods. Finally, as a consequence of using static analysis, this approach will miss out on dynamically included scripts [NIK+12]. Thus, our approach will provide a lower bound on the prevalence of web bot detection in this respect.

property	Chrome	Selenium WD. Chrome (HL)
userAgent	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) **Chrome**/ 64.0.3282.140 Safari/537.36	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) **HeadlessChrome**/ 64.0.3282.140 Safari/537.36
resolution	1920x975	640x480
available Resolution language inconsistencies	1855x951 False	640x480 True
canvas fp	-	Deviates
plugins	Chrome PDF Plugin::Portable Document Format::application/x-google-chrome-pdf pdf,...	None
MIME types	{"0":{},"1":{},"2":{},"3":{},...}	{}
request headers	"content-length":"**65515**", "user-agent":"Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/ 537.36 (KHTML, like Gecko) **Chrome**/64.0.3282.140 Safari/537.36", **"accept-language":"en-US,en;q=0.9"**	"content-length":"**64980**", "user-agent":"Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/ 537.36 (KHTML, like Gecko) **HeadlessChrome**/64.0.3282.140 Safari/537.36"
window keys	-	Missing: **chrome**, **attr**
document keys	-	Added: **$cdc_asdjflasutopfhvcZLmcfl_**
document Elements	-	Additional nodes in document tree: #myId{visibility:visible}", "childNodes":[]}]},{"nodeType":3, "nodeName":"#text",...

Table 7.4: Deviations between headless Selenium + ChromeDriver and Chrome. The resulting fingerprint surface is marked in bold

7.5.2 Patterns to detect web bot detectors

To determine if a website is using web bot detection, we check whether it accesses the fingerprint surface. We do this by checking whether the client-side JavaScript of the website includes patterns that are unique to an individual bot's fingerprint surface. We derived these patterns as follows: firstly, from the the determined fingerprint surfaces, secondly, from the reverse analysis. With these we executed preliminary runs of the scanner, which resulted in more candidate scripts. The third source of patterns stems from new scripts identified in this stage. Table 7.5 lists the used patterns. Patterns derived from reverse analysis of the Distil bot detector are marked as 'RA', while patterns that emerged from the gathered fingerprint surfaces are marked as 'FP'. Finally, later identified web bot detector scripts are marked as 'RA2'. For all patterns where it is clear which web bot they detect, this is indicated in the table. By construction, this is the case for all fingerprint surface-derived patterns. However, not all patterns from the various reverse analysed scripts could as readily be related to specific web bots. These are marked as '?' in the column *Detects* in Table 7.5.

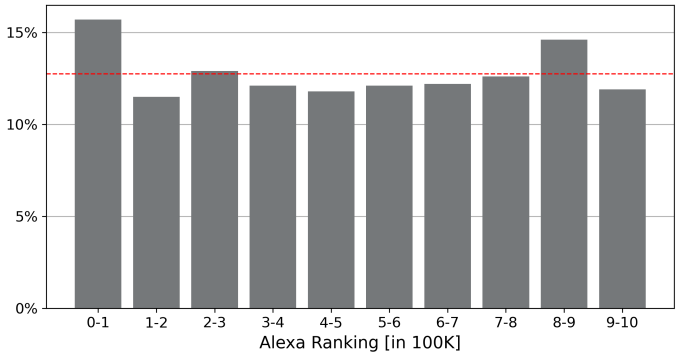

Figure 7.2: Fraction of web bot detectors within the Alexa Top 1M

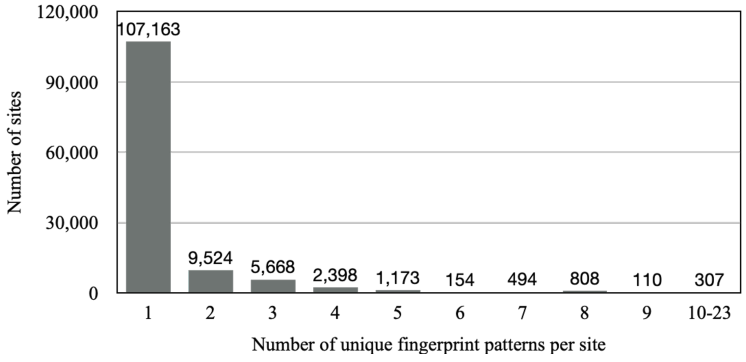

Figure 7.3: Number of unique hits per website. Each pattern is counted once per site

pattern	source	detects
”webdriver”,’webdriver’, \.webdriver(?![a-zA-z-])	FP	IE + Sel. WD
__webdriver_script_fn	FP	IE + Sel. WD
_Selenium_IDE_Recorder	FP	FireFox Selenium IDE
PhantomJS(?![a-zA-z-])	FP	PhantomJS
_phantom(?![a-zA-z-])	FP	PhantomJS
callPhantom	FP	PhantomJS
HeadlessChrome	FP	Chrome + Chromium
__nightmare	FP	PhantomJS
__IE_DEVTOOLBAR_CONSOLE _EVAL_ERROR	FP	InternetExplorer
__IE_DEVTOOLBAR_CONSOLE _EVAL_ERRORCODE	FP	InternetExplorer
$cdc	FP	Chrome* WebDriver
webdriver_evaluate	RA	FX-Driver
fxdriver_evaluate	RA	Selenium
domAutomation	RA	Headless Chrome
_selenium	RA	?
selenium_evaluate	RA	?
webdriver_script_func	RA	?
selenium_unwrapped	RA	?
driver_unwrapped	RA	?
webdriver_unwrapped	RA	?
driver_evaluate	RA	?
fxdriver_unwrapped	RA	?
Sequentum	RA	?
callSelenium	RA	?
script_function	RA	?
$wdc	RA2	Selendroid
webdriver-evaluate	RA2	FX-Driver
domAutomationController	RA2	Headless Chrome
__phantomas(?![a-zA-z-])	RA2	PhantomJS
[’PhantomJS(?![a-zA-z-])’,’botPattern’]	RA2	PhantomJS

Chrome*: Chrome and Chromium.
FP: From fingerprint surface.
RA: From reverse analysis of the Distil script.
RA2: From the reverse analysis of later identified scripts.

Table 7.5: Web bot detector patterns derived from our reverse analysis and fingerprint surfaces

	patterns	# sites	validation		
			sample size	FP	FP%
1	`PhantomJS(?![a-zA-z-])`	115,940	383	4	1%
2	`callPhantom`	11,759	373	0	0%
3	`_phantom(?![a-zA-z-])`	11,747	372	15	4%
4	`HeadlessChrome`	10,279	371	0	0%
5	`"webdriver", 'webdriver', \.webdriver(?![a-zA-z-])`	8,512	368	2	1%
6	`webdriver-evaluate`	5,441	all scripts	0	0%
7	`domAutomation`	2,238	328	0	0%
8	`__phantomas(?![a-zA-z-])`	2,123	317	0	0%
9	`domAutomationController`	1,852	all scripts	0	0%
10	`__webdriver_script_fn`	1,499	306	0	0%
11	`Sequentum`	1,479	all scripts	0	0%
12	`$cdc`	1,251	all scripts	19	2%
13	`$[a-z]dc_`	1,240	all scripts	8	1%
14	`_selenium`	636	240	86	36%
15	`_Selenium_IDE_Recorder`	339	all scripts	0	0%
16	`driver_unwrapped`	318	all scripts	0	0%
17	`fxdriver_unwrapped`	318	all scripts	0	0%
18	`driver_evaluate`	316	all scripts	0	0%
19	`webdriver_evaluate`	315	all scripts	0	0%
20	`selenium_unwrapped`	307	all scripts	0	0%
21	`webdriver_unwrapped`	307	all scripts	0	0%
22	`selenium_evaluate`	306	all scripts	0	0%
23	`__nightmare`	296	all scripts	0	0%
24	`callSelenium`	296	all scripts	0	0%
25	`webdriver_script_func`	295	all scripts	0	0%
26	`webdriver_script_function`	293	all scripts	0	0%
27	`fxdriver_evaluate`	267	all scripts	0	0%
28	`$wdc`	47	all scripts	20	43%
29	`PhantomJS(?![a-zA-z-]) && botPattern`	31	all scripts	0	0%

Table 7.6: Pattern matches within the Alexa Top 1M

7.5.3 Results of a 1-Million scan

We deployed our scanner on the Alexa Top 1M and found 127,799 sites with scripts that match one or more of our patterns. Except for the Top 100K, these sites are mostly equally distributed. In the Top 100K, the amount of web bot detection (15.7K sites) is around a quarter higher than for the rest, which averages to 12.7K sites using detection per 100K sites (see the distribution in Figure 7.2).

Many of these sites employ PhantomJS-detection. In Table 7.6, we see that out of the 180,065 matches to the pattern list, the top three patterns were all PhantomJS-related and together accounted for 139,446 hits. When all PhantomJS-related patterns are grouped, we find that 93.76% of the scripts in which we found web bot detection, contains one or more of these patterns.

While less prevalent, detection of other web bots does occur. The next most popular patterns are related to WebDriver (1.31% of sites in Alexa Top 1M), Selenium (1.34%), and Chrome in headless mode (0.99%). The other patterns were seldomly encountered, none of them on more than 0.2% of sites.

We also investigated how many distinct patterns occurred in detector scripts (Fig-

ure 7.3). Most sites only triggered one pattern, while 96% of these match the pattern `PhantomJS(?![a-zA-z-])`". This suggests that simple PhantomJS checks are relatively common, while actual bot detection using client-side detection is rare. The highest number of unique patterns found on a site was 23.

7.5.4 Validation

In order to validate the correctness of our results, we check if there are non-bot detectors among our collection of bot detector scripts, so called false positives. To confirm a script is a bot detector we perform code reviews. A script is marked as confirmed if it accesses unique web bot properties or values via the DOM. Some detectors separate their detection keywords in a different file, as we encountered that during our reverse analysis in Section 7.2. Therefore, we also interpret these scripts (listing multiple of our patterns) as detectors. Note, our validation is limited to false positives. We do not investigate false negatives (scripts that do perform bot detection, but were not detected): such scripts were not collected.

In a preliminary validation run, we observed that some patterns are more likely to produce false positives than others. Therefore, we assessed false positive rates for the patterns individually by building sets containing all scripts that triggered a specific pattern.

Table 7.6 depicts the results of our validation by showing the set size of validated scripts, number of false positives (FP) and the percentage of false positives per set. For 20 out of 29 patterns, many sites used the exact same script. In these cases, we validated the entire set by reviewing all unique scripts in the set. Any found false positives were weighted accordingly.

For the remaining patterns, the sets of scripts was too diverse to allow full manual validation. Instead, we used random sampling to validate these patterns. To determine the sample size that provides a confidence level of 95%, we used the calculation provided by [Rao05] with a margin of error of 5% and maximum variability (50%). In Table 7.6, we list these patterns together with the taken sample size.

Our validation shows that the patterns *$wdc* and *_selenium* raise a non-negligible number of false positives, though scripts matching these patterns only constitute a tiny portion of our data set. The other patterns are good indicators of web bot detection.

7.6 Cloaking: are some browsers more equal than others?

Finally, we studied whether sites we identified as engaging in bot detection respond differently to web bot visitors. That is: do these websites tailor their response to specific web bots? Note that we manually examine the response to generic web bots, which differs from previous work that investigated cloaking in the context of search engine manipulation [ITK+16; WD05; WSV11].

We first assess the range of visible variations by visiting 20 sites that triggered a high number of patterns. To do so, we visited websites and took screenshots with a manual driven Chrome browser and an automated PhantomJS browser – the most

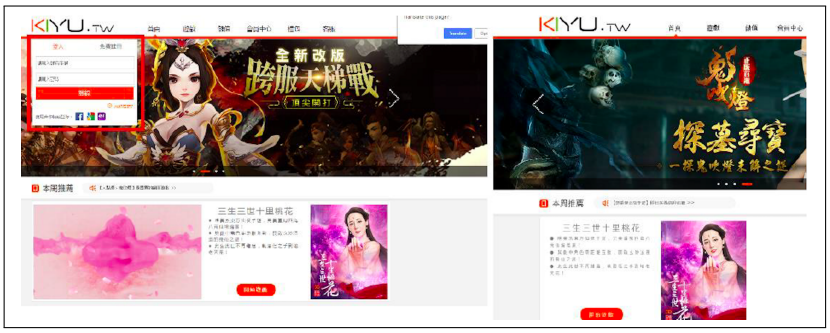

Figure 7.4: Missing login fields on `kiyu.tw`

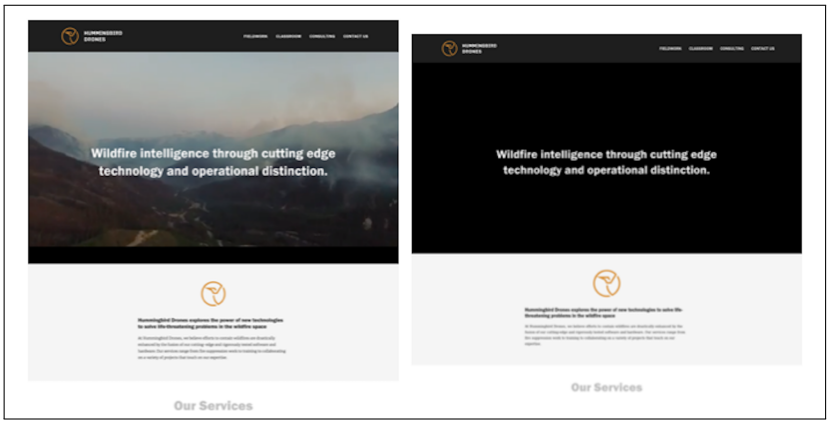

Figure 7.5: Missing video on `hummingbirddrones.ca`

detected and most detectable automation framework in our study. We repeated this five times to exclude other causes, such as content updates. We found four types of deviating responses: CAPTCHAs (3 sites), responses of being blocked (1 site), connection cancellation or content not displayed (1 site) and different content (12 sites).

The differences in content concerned page layout (2 sites), videos that do not load (3 sites), missing ads (9 sites) and missing elements (1 site). We found that these deviations are highly likely to be caused by bot detection, e.g., one site in our set does not display login elements to web bots (see Figure 7.4). In contrast, deviations such as malformed page layouts may be a result of PhantomJS' rendering engine.

We found that sites with missing videos use scripts to serve the videos by wistia.com (cf., Figure 7.5). These scripts include code to detect PhantomJS. We therefore believe the lack of video to be due to web bot detection, though we cannot be certain without reverse engineering these scripts fully.

Lastly, we explored how often deviations due to bot detection occur. In addition to PhantomJS, we added a Selenium-driven Chrome browser. We randomly selected 108 sites out of our set of detectors. Each site was visited once manually and 5 times with each bot, using different machines and IPs. By comparing the resulting screenshots we found deviations on 50 sites. From these deviations, we removed every

observation (e.g., deformed layouts and inconsistent results over multiple visits) that we could not clearly relate to web bot detection. This results in 29 websites where we interpret deviations as a cause of web bot detection.

We found 10 websites that do not display the main page or show error messages to web bots. CAPTCHAs were shown on 2 sites (see Figure 7.7). We further encountered missing elements on 2 sites and videos failed to load on 4 sites. Lastly, 15 sites served less ads (see Figure 7.6). Overall, deviations appeared more often in PhantomJS (24) than in Selenium-driven Chrome browsers (14).

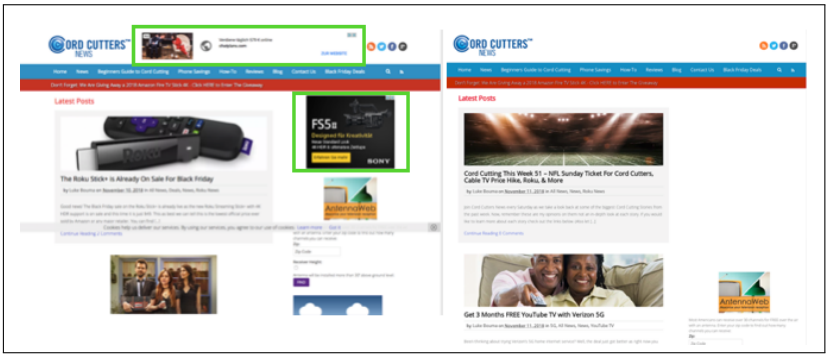

Figure 7.6: Missing ads on `cordcuttersnews.com`

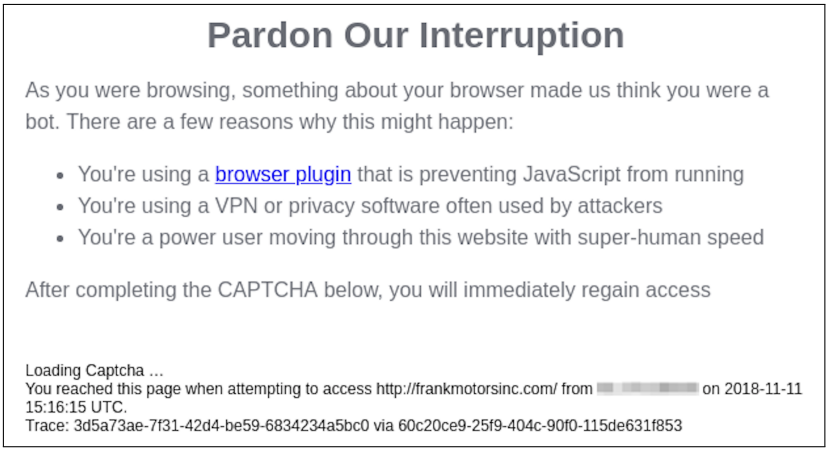

Figure 7.7: Blockage and loading of a CAPTCHA on `frankmotorsinc.com`

7.7 Conclusions

The detection of web bots is crucial to protect websites against malicious bots. At the same time, it affects automated measurements of the web. This raises the question of how reliable such measurements are. Determining how many websites use web bot

detection puts an upper bound on how many websites may respond differently to web bots than to regular browsers.

This study explored how prevalent client-side web bot detection is. We reverse engineered a commercial client-side web bot detection script, and found that it partially relied on browser fingerprinting. Leveraging this finding, we set out to determine the unique parts of the browser fingerprint of various web bots: their *fingerprint surface*. To this end, we grouped browsers into families as determined by their layout and rendering engines. Differences between members of the same family then constituted the fingerprint surface. We determined the fingerprint surface of 14 web bots. We found PhantomJS in particular to stand out: it has many features by which it can be detected.

We translated the fingerprint surfaces into patterns to look for in JavaScript source code, and added additional patterns from the reverse analysis and common best practices. We then developed a scanner built upon OpenWPM to scan the JavaScript source of the main page of all websites in the Alexa Top 1M. We found that over 12% of websites detect PhantomJS. Other web bots are detected less frequently, but browser automation frameworks Selenium, WebDriver and Chrome Headless are each detected on about 1% of the sites.

Lastly, we performed a qualitative investigation whether web bot detection leads to a different web page. We found that indeed, some browsers are more equal than others: CAPTCHAs, blocked responses, and different content occur. In a further experiment, we attribute at least 29 out of 108 encountered differences.

Chapter 8

Generic Detection of Web Bots

Web bot defences can improve coverage by combining multiple methods. So far, three methods have been used: browser fingerprinting, order of site traversal, and aspects of page interaction. While site traversal depends on the study being executed, the other two aspects can be controlled in a generic fashion. However, browser fingerprinting necessitates prior knowledge of the used technology in the visiting bot. Contrasting page interactions are free of this limitation and have the potential to detect any bot that interacts on a page. To date, little is known to what extent automation frameworks are prone to detection via this method.

In this chapter, we provide an initial investigation of an interaction API of the web automation framework Selenium to explore how it differs from human interaction. We incorporate the latter results into HLISA, an API that simulates human interaction. Finally, we discuss the conceptual arms race between simulators and detectors and find that conceptually, detecting HLISA requires modelling human interaction.

This chapter is based on the following publication:

HLISA: towards a more reliable measurement tool. Goßen, Daniel and Jonker, Hugo and Karsch, Stefan and Krumnow, Benjamin and Roefs, David. *In Proc. 21st ACM Internet Measurement Conference (IMC'21)*, DOI: 10.1145/3487552.3487843, 2021, [IMC21].

8.1 Introduction

Web measurements that account for certain website functionality (cf., [MADWeb20; DIP20; MADWeb23]) or aim to simulate human users typically interact with websites [CHP+19; UDH+20; SAH+22]. This interaction includes opening subpages, moving the cursor to trigger hover events, clicking page elements to activate them, and typing to fill out forms. However, page interaction is a possible data source to distinguish bots from web users [CGK+13]. Bot detection can undermine the fundamental assumption of web studies: that the measuring tool used encounters the same Web as regular browsers would.

Serving different content hinges on recognising web bots as such. Three avenues for web bot detection have been identified: browser fingerprinting [ESORICS19; VRR+20]), site traversal (cf., [TK02]), and interaction characteristics (cf., [CGK+13]). Web studies that want to minimise errors due to web bot detection should account for all three aspects. Crucially, mitigating site traversal – the path an automated browser takes over a website – cannot be solved generically, as such paths depend on the study being executed. However, neither browser fingerprint nor interaction characteristics are (typically) study-dependent. Both aspects can thus be generically addressed.

In this chapter, we investigate these two aspects. More particularly, previous studies have found what properties to change about the browser fingerprint: eliminating the `webdriver` property suffices for most contemporary detection approaches, while methods to determine the full fingerprint needed are available [ESORICS19; SLG19]. However, *how* to alter the browser fingerprint has not been as comprehensibly studied. We investigate various ways to spoof aspects of a browser fingerprint using JavaScript and elucidate the side effects inherent in each approach. Secondly, previous studies and online advice usually focused on changing one specific form of interaction to reduce the distinctiveness of its automation. In contrast, we offer a comprehensive study on identifying and reducing distinctiveness for all types of user interaction observable via JavaScript.

Contributions. The contributions of this work are:

- We compare known means to spoof identifiable properties in web bot frameworks. We are the first to investigate side effects that allow the detection of a spoofing attempt. Moreover, we test the effectiveness of the best method in the wild.

- We experimentally establish the recognisability of the Selenium interaction API and provide a proof-of-concept interaction library, HLISA, to address the identified shortcomings.

- We model the action-reaction cycle between detectors and simulators as an arms race. Our model clarifies a detector's and simulator's limitations and how to escalate their respective capabilities. Moreover, the model enables a legal analysis of detection approaches vs privacy regulations.

Ethical considerations. For the design and execution of our experiments as well as the development of HLISA, we considered multiple ethical issues and integrated mitigations when necessary.

Involvement of human participants. The experiments in this work involving human subjects were extremely limited in scope. These experiments aimed not to establish an average of generic human behaviour but to contrast Selenium's interaction with that of an individual human. As such, we limited the involved human participants to ourselves. We asked the ethical advisory board for advice on whether approval by the IRB was necessary. The advisory board agreed that in this case, for the specific, limited amount of data gathered from the authors themselves, a full, formal ethical review process was not needed.

Unintended negative secondary effects (dual use). Though our work aims to improve the accuracy of tooling used in scientific web studies, our work may have secondary effects. As our work helps to make web bots less distinguishable from regular visitors, it may be leveraged by nefarious bots and benign bots. In fact, malicious web bot campaigns already incorporate simulated human interaction that circumvents fraud detection (e.g., Methbot [Whi16]). Methbot's interaction capabilities seem to be at least as good as those of HLISA, so HLISA is not extending the capabilities of criminals. This leaves benign uses of HLISA: for researchers to augment their scraping bots; and for websites to augment their bot defences (by incorporating behavioural bot detection trained against HLISA).

With these effects in mind, we believe the beneficial effects to outweigh the potential negative consequences of our research.

Limited applicability of the model. HLISA's interaction model relies on the data collected from white, male, Western-European, highly educated subjects studying computer science. Obviously, these subjects are not representative of the world's population. As such, we caution against using HLISA as-is for other purposes (e.g., usability testing).

Outline. The structure of this chapter is as follow: we contrast multiple approaches in JavaScript to hide a web bots' identifiable properties (Section 8.2). Then, we investigate produced interactions by Selenium and OpenWPM for recognisable patterns (Section 8.3). Following this, we present our library HLISA (Section 8.4). In the final part, we provide a theoretical model of the simulator-detector arms race (Section 8.5) and present our final remarks (Section 8.6).

8.2 Evading fingerprint-based detection

Browser fingerprinting allows websites to detect bots without any interaction on a page. Hence, to prevent bot detection, the first defence is to hide a web bot's identifiable properties.

There are two approaches to changing browser properties: modifying the browser's source code and overriding properties at run time (e.g., via JavaScript). Both have their merits and disadvantages. Overriding based on the built-in functionality of the JavaScript API can be applied dynamically, even for properties created at run time. Furthermore, it is easy to deploy, as JavaScript code is directly injected into a web

Table 8.1: Detectable side effects by spoofing methods

side effect	spoofing method 1	2	3	4
incorrect order of `navigator` properties	×	×		
modified `navigator._length`	×	×		
new `Object.keys(navigator)`	×	×		
defined `navigator.__proto__.webdriver`			×	
unnamed `window.navigator` functions				×

page and is cross-platform compatible. One downside is that side effects may occur due to JavaScript quirks. Moreover, such modifications can break websites, as already found for some privacy extensions [Cro18].

In contrast, browser-level patches of properties avoid the introduction of such side effects. However, adjusting the browser's source code adds considerable overhead. First, adjustments must be maintained for newly appearing browser versions. Second, the browser build process may be a challenge for some users. Lastly, a browser-level implementation binds a solution to a single platform. However, it needs to be determined if JavaScript-level patching can be made without thwarting web measurements.

In the remainder of this section, we explore approaches in JavaScript for property spoofing. We use our insights to build an extension that can hide detectable properties for OpenWPM, a popular framework to measure web privacy and security aspects (see Section 9.2). Finally, we test our implementation on 1,000 websites.

8.2.1 JavaScript-based spoofing methods

To select spoofing methods suitable for our evaluation, we consider approaches used earlier in the literature and approaches adopted by popular browser extensions. In more detail, we conducted source code reviews of spoofing extensions available for Firefox [Lin20; Ray20; Sch20; ser20; ner20], academic work to fight browser fingerprinting [FZW15; TJM15; CY19; NJL15] and research to block ads [SRM+17]. This led us to the following methods:

1. **defineProperty** is a built-in function of JavaScript objects to set or alter an object's property directly.

2. **__defineGetter__** overrides a `getter`-function allowing us to return a specific value without changing it. Note that Mozilla deprecated this function.

3. **setPrototypeOf** sets a new prototype for an object which provides control over the object's properties.

4. **Proxy objects** allow to re-define the behaviour of an object by wrapping it with a proxy object.

We tested each method to spoof `navigator.webdriver` property[1] to return *false* within Firefox. Note that this is an easy-to-use property available by convention and

[1] `https://www.w3.org/TR/webdriver/`

plays a crucial role in identifying WebDriver-controlled user agents. [VRR+20]. To check for the occurrence of side effects of each method, we use JavaScript template attacks by Schwarz et al. [SLG19] and the modified fingerprint library that we developed in Section 7.4.

Table 8.1 lists the four evaluated methods along with their side effects. Interestingly, none of the previously applied methods was side-effect free in our measurement. For methods 1 and 2, we observe that each attempt to spoof a property increments the `navigator.length` property. Spoofing the length property in this manner is insufficient, as its original value remains in the prototype chain. In addition, we find a change in the order of items when iterating through the navigator's properties. This reveals which property has been overwritten. In a regular Firefox browser, the `webdriver` property is enumerable but disappears from the listing when calling `Object.keys(navigator)` in our overwritten version. However, it is possible to remedy this by setting the enumerable property to *true*. The third method does not include the previous method's side effects but is inherently detectable. In regular Firefox, the chain `__proto__` is not defined for the `webdriver` property but needs to be set in order to execute this method. We find that wrapping the navigator object is detectable by calling the `toString` function for the last approach. As shown in Listing 8.1, the result of this overriding leads to missing function names.

```
1  //Call of a toString function of a built-in method
2  window.navigator.toString.toString();
3
4  // Output in a regular Firefox browser
5   "function toString() {
6     [native code]
7    }"
8
9  // Output after shadowing methods via proxy objects
10  "function () {
11    [native code]
12   }"
```

Listing 8.1: Detectability of JavaScript spoofing based on proxy objects.

We conclude that JavaScript proxy objects appear to be most suitable for spoofing. While an adversarial website could spot a wrapped navigator object, it does not know what property was changed when applying this approach to multiple properties. Further, benign web users may apply the same techniques through extensions, e.g., userAgent spoofers, privacy extensions, and such. Still, the effectiveness of this approach must be evaluated on real-world websites.

8.2.2 Evaluation

We developed a browser extension to spoof the *webdriver* property in OpenWPM clients based on our selected method. To evaluate our extension in the field, we run OpenWPM with and without our extension on two different machines. We use a consistent setup with OpenWPM v.0.13.0 and run the Firefox browsers in headful mode. We then let both machines visit the same set of websites simultaneously. Our website set consists of a random selection of 1,000 sites taken from the Top 10K websites of the Tranco list [LVT+19]. As our experiment could be influenced by web dynamics (bidding processes, content updates, etc.) as well as blocking through

suspicious IP ranges (e.g., university or cloud-based IP addresses [ITK+16; ZBO+20]), we take precautions. First, we use different residential IP addresses, and second, we run eight browser instances simultaneously per machine. This also helps us to improve robustness, as we can check the consistency of effects over multiple bots.

Incidence of blocking. We check if our extension mitigates bot detection while not breaking websites. To measure incidences of detection, we review screenshots and count the occurrence of blocking pages, CAPTCHAs, visible error messages and HTTP response status codes that occur only for one machine. In addition, we evaluate if there is missing content (such as ads). We choose to use visual responses as these allow definitive attribution to bot detection, while other measurable aspects (e.g., cookies or HTTP traffic) do not. Not all websites with bot detectors react visually to automated visitors [JSS+21]. As such, this check is only an approximation to determine the effect of using the spoofing implementation.

Table 8.2 breaks down our results of screenshot evaluation separated into reached sites and successful visits. We see a low number of sites with visible signs of bot detection. All observed differences combined occur on only 16 (1.7%) sites for the vanilla OpenWPM client. We find that spoofing reduces this effect significantly. In fact, we see only one site that deploys blocking against our extended OpenWPM version for a smaller subset of visits.

To identify blocking at HTTP level, we look at status codes in HTTP responses. We separated these by first and third-party responses. We further use Wilcoxon Matched-Pairs signed-Rank Test with a confidence interval of 95% to test for significance. Our results show only a notable difference in first-party errors, with a significant decrease (p-value = 0.004) when using our extension. Figure 8.1 depicts the occurrence of error-related HTTP responses, which shows that this decrease is mainly due to responses with 403 (forbidden) and 503 (service unavailable). Both status codes can be related to the effect of bot detection.

Incidence of website breakage. Lastly, we look for deformed layouts, frozen elements, missing content and HTTP errors to spot if any website breakage occurred. Interestingly, we identified one site with a deformed layout and one site with an ever-loading video element, which hints at compatibility issues on these sites. Unfortunately, while the effect persisted when running the experiment on another machine, we failed to identify the root cause for these breaks.

Table 8.2: Results from the screenshot evaluation

response	sites		visits	
	(1)	(2)	(1)	(2)
total	**921**	**921**	**7,230**	**7,221**
missing ads	7	3	56	10
– no ads	5	1	40	4
– less ads	2	2	16	6
blocking/CAPTCHAs	8	1	49	3
frozen video element(s)	1	0	8	0

Results for crawler OpenWPM (1) and OpenWPM+extension (2).

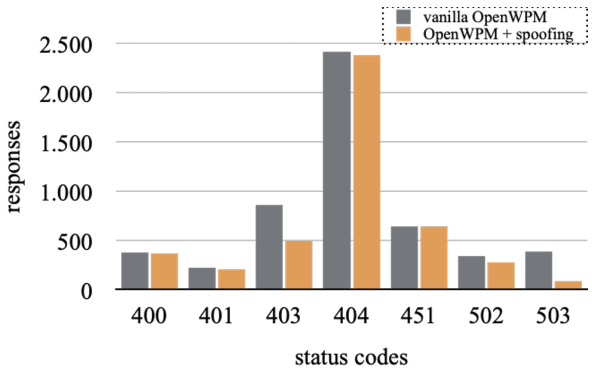

Figure 8.1: HTTP errors with more than 100 occurrences grouped by status code

8.3 Recognising generated interaction

Our goal is to identify how page interaction generated by an automation framework differs from non-automated clients. A web page can monitor a web client's interactions through JavaScript events. We take the website perspective by setting up an experiment where we measure JavaScript interaction events of web automation frameworks and humans.

Again, we start with OpenWPM. Nevertheless, OpenWPM simply exposes the Selenium interaction API, wherefore measurements with either of these tools lead to the same outcome. The only difference to Selenium is an opt-in bot detection mitigation feature offered in OpenWPM. According to the developers, the feature does not provide a thorough simulation of human behaviour [BC21]. Still, a study by Ahmed et al. [ADZ+20] reports a lower occurrence of CAPTCHAs due to the mitigation feature turned on. We investigate the strength of this protection and if specific characteristics make OpenWPM's bot detection mitigation detectable.

8.3.1 Measuring interaction

We performed several experiments to compare the interaction of Selenium with that of a regular human (ourselves). For that, we distinguish between fine-grained and course-grained actions. A fine-grained action is a single operation, such as a mouse button press. In contrast, a coarse-grained action consists of a chain of fine-grained actions. For example, clicking an element on a page could consist of three steps: moving the cursor, pressing the mouse button, and releasing it. We based our investigation on coarse-grained actions that are essential for typical web browsing activities (typing, clicking, mouse movement and scrolling).

Human interaction. We built a website that uses JavaScript to record events. For each interaction type, our site asks the user to perform simple tasks.

We took measurements on typing by letting the user type a given text of 100 characters. Our page recorded vital press and key release events, including the timestamp. From this, we derived dwell and flight times for a user's keystrokes (dwell time

denotes the time between a key press and a key release; flight time is the time between a key release and a key press).

For mouse movement, we record the cursor coordinates and timestamps for each `onmousemove` event. We use this data to monitor the cursor's distance, slope and speed. In addition, we create a visual representation of the path taken. The site instructed the participant to click two distant elements in a specific order so that the interaction starts and ends at similar positions.

We created a moving element for mouse clicks to collect data from various angles. The element relocates every time it is clicked. Our human participant repeated this task 100 times, where we recorded the dwell time and the position of each click.

Last, we created a sufficient long page (height 30K pixels) to record scrolling events. The task was to scroll via the mouse wheel from top to bottom at a comfortable pace. This provides realistic data on one scrolling method; other methods are not considered.

Selenium's interaction. To analyse the behaviour exhibited by Selenium's interaction API, we first determined how to measure any interaction. OpenWPM uses the Firefox browser, which offers 57 events (see Appendix B.1) related to or triggered by interaction. Many of these provide overlapping information. The following set of 10 events together cover all interaction information available to a web page:

- **Mouse movements:**
  - `mousemove`
- **Mouse clicking:**
  - `dblclick`
  - `mousedown`
  - `mouseup`
- **Scrolling:**
  - `scroll`
  - `wheel`
- **Typing:**
  - `keydown`
  - `keyup`
- **Touch:**
  - `touchstart`
  - `touchend`
- **Losing/gaining focus:**
  - `visibilitychange`
  - `blur`
  - `focus`

Interestingly, the granularity of events varies. In particular, the granularity of **mouse movement** events can vary. Moreover, we did not find a correlation between the number of events fired per second and mouse movement speed. All in all, we found Firefox's event API too coarse to register every detail of normal mouse movement. In contrast, events for all other categories provide a more comprehensive view. For example, the granularity for **typing** events is 1 ms. Interestingly, for **mouse clicking**, Firefox asks its environment what the maximum interval is to consider two consecutive clicks a double click. For Windows, this defaults to 500 ms; in testing Selenium, we found a maximum interval of 600 ms.

Scroll-related events can be triggered in many ways, including: mouse wheel, trackpad scrolling, scroll bar, arrow keys, using find, URL anchors, and auto-scrolling. This wide array of origins, each causing a different amount of scrolling, significantly restricts the effectiveness of using this type of interaction to distinguish bots from human visitors. Thus, even though the amount scrolled by a scroll-wheel 'click' is

fixed (57 pixels in our setup), different scrolling amounts can and will occur in normal use. The absence of a `wheel` event or a different amount of scroll distance thus does not suffice to distinguish a web bot from humans. **Touch** movement can also indicate a touchpad, and, as such, is included in `mousemove`. Finally, **focus** events can be triggered by Selenium also after minimising a headful browser. Minimising causes a `visibilitychange` event, after which no further interaction should occur. This should be addressed in the experiment design.

8.3.2 Recognising Selenium's interaction

To determine in what respect the behaviour of Selenium is distinctive, we challenged visitors to perform specific tasks on a test site and measure aspects of the resulting behaviour. Our experiment shows that Selenium's speed and precision far outstrip human capabilities. We tested four aspects of Selenium's behaviour via simple web pages: typing, mouse movement, scrolling via touchpad/mouse wheel, and clicking. We visited each of these pages with Selenium and manually by ourselves. While the latter does not constitute a thorough measurement of human behaviour, this was sufficient to identify surprisingly large differences.

- **Typing.** For typing, we measured dwell and flight time; see Table 8.3. Selenium types much faster than our human test subject.
- **Scrolling.** With respect to scrolling, we noted that human scrolling (irrespective of the input device used) generates many events, each of which with a limited number of pixels to scroll (95% within [1, 50]). A bot-invoked scroll generates one event with the number of pixels requested – which can easily be beyond the expected human range.
- **Clicking.** We measured clicks on elements based on the location and dwell time. Table 8.4) lists exemplarily three clicks on a non-moving target by Selenium and a human user. We observe that Selenium clicks precisely in the centre of an element every single time. In contrast, human clicking is typically off-centre by at least a few pixels in both x- and y-directions. Moreover, the average human dwell time is significantly longer (100–160 ms) compared to Selenium (0.15–0.3 ms). This remains true even for the fastest human dwell times we recorded (7 ms on a touchpad). Finally, humans typically move the mouse while clicking (e.g., when double clicking); Selenium does not when using the `click` function.
- **Mouse movement.** Selenium starts moving at the top left corner (0, 0). This happens each time the browser is freshly started. Moreover, the movement is always in a straight line. In contrast, all human mouse movements we recorded were more curved and sometimes contained errors (e.g., overshooting the target, clicking next to an element, and such).

OpenWPM's bot detection mitigation. OpenWPM's feature produces random interaction on a page, after loading it. It encompasses three steps which always occur in the very same order: random mouse movement, scrolling down the page within a random distance, add a random timeout. While the latter does not lead to measurable

Table 8.3: Bot vs. human typing characteristics. Dwell and flight time in ms.

	Bot		**Human**	
	dwell	flight time	dwell	flight time
avg	0.76	0.41	90.73	86.77
max	2.58	0.80	121.60	395.76
min	0.34	0.16	11.52	10.18
std	0.48	0.17	32.52	37.12

Table 8.4: Bot vs. human click characteristics

Bot			**Human, mouse**			**Human, touchpad**		
x	y	dwell	x	y	dwell	x	y	dwell
45	230	0.22	57	226	135.18	54	232	6.84
45	230	0.18	39	228	110.46	30	233	7.16
45	230	0.22	54	226	103.76	66	231	6.30

Dwell given in milliseconds.
x and y give pixel values on the screen.

events, we focus on scrolling and mouse movement. To that end, we reviewed the source code and set up a page to measure the resulting interaction.

We found that OpenWPM produces many events that move more than 50 pixels when scrolling. In contrast, human user scrolling typically contains very few large distances (beyond 50 pixels) but many small events that move only a few pixels. For mouse movement, OpenWPM uses a handful of random points on the screen. Each intermediate step is once again a straight line, as shown in Figure 8.2. Due to the usage of bot mitigation before any other interaction, the movement starts in the top left corner of the screen. Each move is carried out without timeouts in between or the consideration of speed. All these features will make it detectable when monitoring the slope or acceleration of the movement.

8.4 Improving Selenium's interaction API

We designed and implemented the *Human-Like Interaction Selenium API* (HLISA) to reduce the detectability of any web bot using the Selenium framework (such as OpenWPM). To simulate more human-like interactions, we use the data collected in our experiment (see Section 8.3.1). Our API advances mouse movement, scrolling, clicking and typing. Finally, we collect interaction data on naive approaches to improve Selenium's shortcomings and contrast them with our solution.

Mouse movement. Mouse movement of a human has an initial acceleration, a deceleration near the end of the trajectory, and moves in a jitterish curved trajectory. Selenium's interaction API, in contrast, moves at a uniform speed over a straight line. A naive solution would be to use a straightforward Bézier curve, but as is apparent from Figure 8.3, this is still very artificial. In contrast, HLISA modifies a Bézier curve by starting with acceleration and ending with deceleration over a jittery curve.

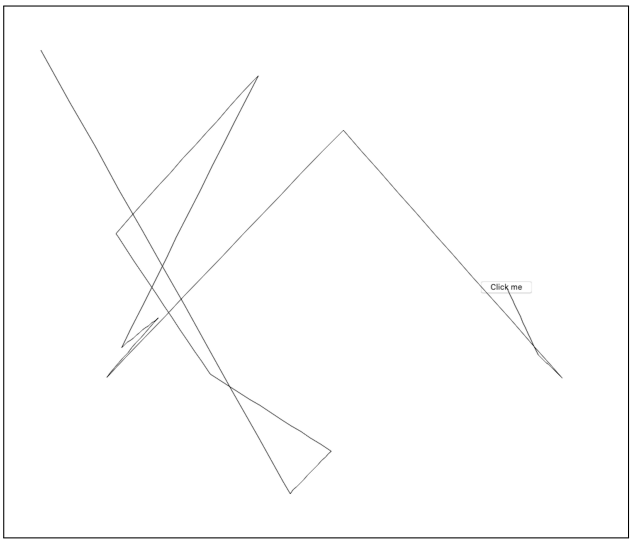

Figure 8.2: OpenWPM's anti-bot detection mouse movement

We use the speed, acceleration and jitter of the mouse movement observed in our experiment (see Section 8.3.1) as a baseline.

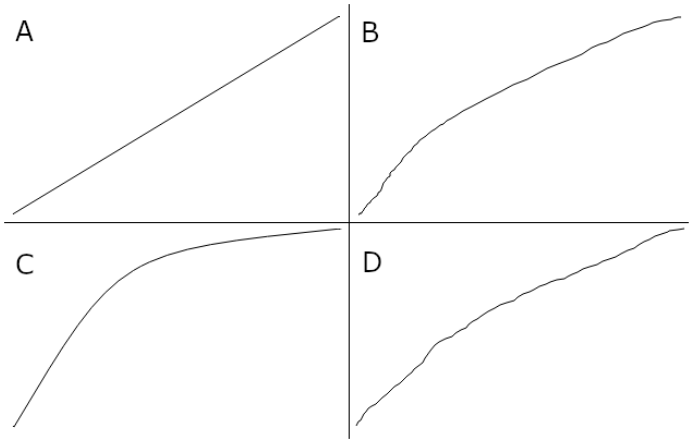

Figure 8.3: Cursor trajectories for: (A) Selenium, (B) human, (C) naive solution, (D) HLISA

Mouse clicks (Figure 8.4). Selenium clicks perfectly in the centre of the HTML element. In contrast, clicks by humans are much more distributed but hardly ever in the centre. A naive approach would be to randomise the click location, e.g., using a uniform distribution (bottom left in Figure 8.4). While this does improve over Selenium's default behaviour, it generates clicks in places humans never reach. In

contrast, HLISA uses a normal distribution with parameters drawn from our experiment.

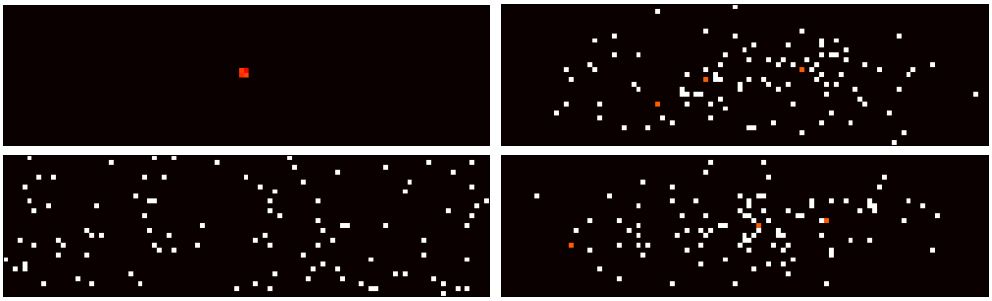

Figure 8.4: Experiment: distribution of mouse clicks of (top left) Selenium, (top right) humans, (bottom left) naive solution, (bottom right) HLISA.

Scrolling. Selenium does not offer an API for scrolling; its default method lacks mouse wheel events and can scroll arbitrary long distances in one scroll event in contrast to human scrolling via the mouse wheel. However, there is a plethora of other human interactions that trigger scrolling, including scroll bar, anchor links, arrow keys, space bar, search functionality, and Firefox's auto-scroll. As such, detecting bots based on scrolling is challenging, and no obvious, naive solution that addresses Selenium's shortcomings. Nevertheless, HLISA extends the Selenium API with a function to simulate scrolling: it uses the default scroll distance (57 pixels for a mouse wheel), a normal distribution to incorporate short breaks, and it incorporates a slightly longer break to account for moving one's finger to continue scrolling the mouse wheel.

Key presses. Key presses of Selenium have a negligible dwell time. Moreover, overall typing speed is inhumanly fast (13,333 characters per minute) and flawless. In our experiments, we observed that fast typing with ten fingers (600 characters per minute) can cause interleaving key presses, i.e., sometimes a key is only released when a different key has already been pressed.

Moreover, while humans need to press modifier keys to press characters like capital letters, Selenium can input any character that exists without pressing additional modifier keys. By monitoring the usage of modifier keys, detectors can infer the keyboard layout, which can be used for static fingerprinting purposes. In contrast, HLISA ensures dwell time is random, drawn from a normal distribution parametrised with values found in our experiment. In addition, HLISA simulates a key press for the Shift key when needed. Finally, HLISA incorporates the timing of various pauses from the work of Alves et al. [ACdS+07]. Hence, HLISA takes into account a variety of timings of events, such as pauses after opening or closing a sentence, writing a new word, using commas, and many more.

Implementation and deployment. HLISA triggers events by calling fine-grained interaction functions (e.g., `move_to_offset(x,y)`, `key_down()`, and `key_up()`) of the

```
# Importing the HLISA library
from HLISA.hlisa_action_chains import HLISA_ActionChains

# Creating an ActionChain with HLISA
ac = HLISA_ActionChains(webdriver)

# Selecting an element
element = driver.find_element_by_id('text_area')

# Adding mouse movement and typing with HLISA
ac.move_to_element(element)
ac.send_keys_to_element(element, "Text..")

# Executing a chain
ac.perform()
```

Listing 8.2: Code example for clicking and sending keys to an element with HLISA.

original Selenium API. This makes HLISA resistant to changes in the Selenium source code that do not affect the Selenium API. The default Selenium API enforces a lower bound on the duration of mouse movements that is too high for simulating human interaction. For Selenium versions <4, we change this duration to 50 msec by overriding the internal Selenium function `create_pointer_move()`. This allows us to express human-like mouse movements.

HLISA's API provides the same calls and signatures as in the original Selenium API (as shown in Table B.1 in the Appendix); except for a few additions. This allows developers to integrate HLISA by modifying two lines of code (see red code in Listing 8.2). Hence, HLISA is compatible with all Python projects already using Selenium.

8.4.1 Current limitations of HLISA

HLISA's current implementation is limited in its capability to impersonate human interaction in two general ways. First, some critical aspects of human behaviour transcend individual interaction with a page but concern more generic behavioural aspects that require separate modelling. Whether and to what extent such behaviour should be simulated depends on the specific experiment being conducted and thus should not be integrated into HLISA. For example, human visitors may exhibit non-functional interaction with webpages, such as selecting and deselecting parts of a page without purpose. Such idiosyncrasies of human behaviour cannot be executed independently by an interaction API, as it may interfere with an experiment's purpose. In a similar vein, other aspects of interaction increase distinctiveness, which operators should handle on the level of an experiment outside of any specific interaction API. These include:

- Mouse movement starting at (0,0), which can be solved by moving the mouse prior to loading a page
- Adding random/spontaneous mouse movements
- Misclicking

- Introducing typing errors and more complex typing behaviour such as reformulating sentences, pausing in longer texts and erasing and cancelling input.

While such aspects cannot be delegated to an interaction API, we see ways to improve HLISA's approximation of human interaction. A general caveat is that HLISA currently uses a normal distribution (parameters following from our experiments) to introduce noise in behaviour, while human behaviour is not normally distributed [CGK+13]. Similarly, HLISA is a proof-of-concept interaction API. It accounts for basic measurements of interaction, such as dwell and flight times of clicks and key presses. Advanced measures of interaction (such as adapting mouse movement to target size and shape) go beyond its proof-of-concept nature. For the same reason, HLISA does not account for touch actions.

Table 8.5: Comparison of libraries or code samples to simulate human like behaviour. A '✓' indicates the functionality is present in the library or code sample

functionality	package							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**HLISA**
mouse movement functionality	✓	✓	✓	✓		✓	✓	✓
realistic mouse movement speed		✓	✓	✓		✓		✓
movement accelerates/decellerates		✓	✓	✓				✓
movement shivering		✓		?[a]				✓
curve in movement[b,c]	✓	✓	✓	?[a]		✓		✓
moves to random location in element[b]								✓
click functionality		✓	✓	✓		✓	✓	✓
realistic dwell time[b]				?[a]		✓		✓
simulates accidental right click			✓					
simulates accidental double click			✓					
simulates accidental no click			✓					
scrolling functionality					✓			✓
pause between scroll ticks					✓			✓
pause for finger replacement					✓			✓
realistic scroll distance in tick					✓			✓
keyboard functionality							✓	✓
flight time[b]							✓	✓
dwell time[b]								✓
timings based on data							✓	✓
other features								
Selenium ready	✓				✓			✓

1. A: "Human-like mouse movement": answer using B-spline curves to question on StackOverflow https://stackoverflow.com/a/48690652
2. Pyclick: Python library for mouse movement using Bézier curves https://github.com/patrikoss/pyclick
3. BezMouse: Python tool for mouse movement using Bézier curves, to avoid bot detection in games https://github.com/vincentbavitz/bezmouse
4. Python Human Movements: Python package to simulate human movement https://pypi.org/project/pyHM/
5. Scroller: tool to simulate human scrolling in Selenium https://github.com/hayj/Scroller
6. ClickBot: Java tool to simulate mouse movement and clicks https://github.com/amSangi/ClickBot/
7. Bachelor thesis [Noo19]: incorporates typing rhythm from HCI literature in Java framework

a. The project links to source code that is incomplete

b. Absence of this feature makes interaction obviously artificial

c. Previously required to bypass Google reCaptcha https://stackoverflow.com/a/37220168

8.4.2 HLISA in comparison to other tools

In Table 8.5, we compare HLISA's features with other tools that simulate parts of human interaction. These come from various sources, ranging from tools focusing on browser interaction (e.g., Scroller) to tools focusing on scripting games (e.g., BezMouse). To the best of our knowledge, HLISA is the only tool that can be used ad-hoc with Selenium while delivering comprehensive coverage of the Selenium API.

8.5 An arms race model of interaction

Conceptually, websites (as detectors) and web bots (as simulators) are engaged in an arms race. Analogous to the ad-blocking arms race as modelled by Storey et al. [SRM+17], detectors and simulators can refine their current techniques and escalate the war by introducing more vigorous techniques.

In Figure 8.5, we depict a conceptual model of this arms race. Note that other detection mechanisms (e.g., fingerprinting) will give rise to a similar arms race. Simulators begin by exhibiting unlimited behaviour: unhumanly fast, unhumanly perfect, and able to interact with all elements irrespective of visibility. Detectors can identify bots by detecting these aspects. This can cause simulators to limit their interaction to that which is humanly possible: within human speeds, including noise instead of perfect replayability and accounting for visibility. Either side can refine their techniques further, succeeding in detecting (e.g., detecting artificialness of noise, adding honey elements) or evading detection. In addition, detectors can escalate, by moving from detecting specific aspects of how simulators interact, to detecting deviations from an expected baseline (human interaction). This forces the simulators to move to simulating human interaction. Once again, either side can refine their techniques – in this case, the models on which detection and simulation is based. The next escalation is to recognise that certain interactions are correlated. For example, faster mouse movement may be correlated with higher (or lower) accuracy clicks. Detectors that move to this level will detect simulators that lack such internal consistency in their interactions. Simulators can, of course, adopt such consistency, which will ultimately defeat detection based exclusively on interaction. The detectors can only escalate further by incorporating information beyond page interaction (marked with dotted lines in Figure 8.5), such as browser fingerprints or individual interaction profiles for specific users (e.g., social media sites have sufficient data for this). This requires an enrolment period during which the detector learns the specific individual's interaction patterns. The only way to defeat such detection mechanisms is to move from simulating interaction that is plausibly human; it simulates the specific interaction profile of a specific individual.

Finally, note that privacy regulations such as the EU's GDPR can set a legal limit to how far detectors can evolve. For example, any detection technique that could be used to identify and track a person would likely fall under the GDPR's purview. The top two detection levels focus detection to such an extent, that individual users could be distinguished. That may run afoul of privacy regulations, though a detailed legal analysis is needed to determine the exact legal limits.

In conclusion, we find that the simulators can always beat the detectors by making use of the same models. This works well for the first few detection strategies, as these

are based on generic findings. However, higher up, this becomes more complex: the exact model of consistency needed to satisfy a detector may not be public knowledge. While this complicates matters for the simulators, they have the advantage that detectors must not be too strict or risk barring human visitors entry. Finally, HLISA offers a simulation of human interaction. As such, it is situated at the third level in the hierarchy of Figure 8.5. Thus, consistently defeating HLISA requires tracking consistency of behaviour. We caution that HLISA is not the endgame of this level; several refinements are possible to approach human interaction more closely.

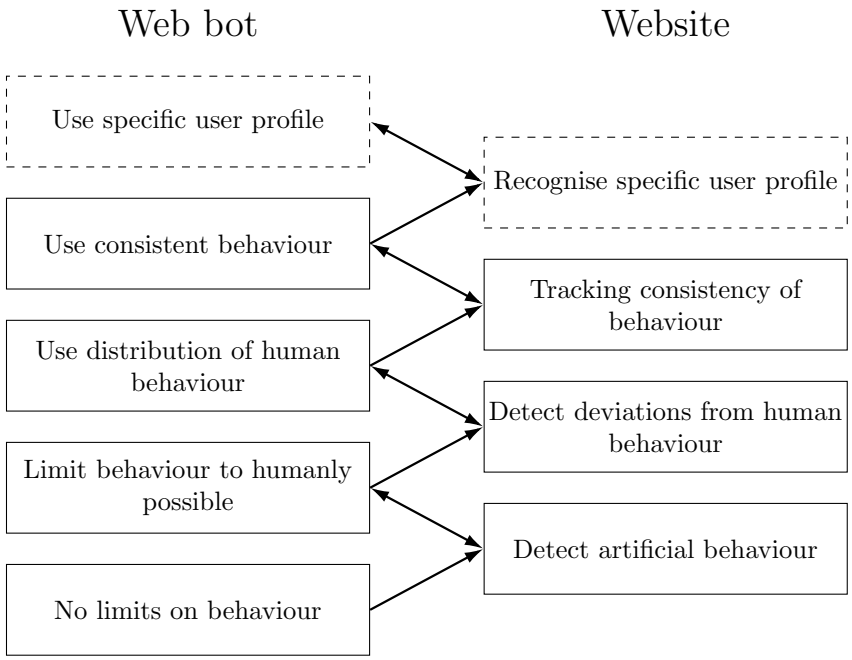

Figure 8.5: A model of the arms race for page interaction

8.6 Conclusions

In this chapter, we (1) investigated the side effects of various methods to alter the browser fingerprint, (2) investigated how page interaction using Selenium's interaction API is recognisably different from human interaction, (3) incorporated our findings on interaction into HLISA, a new interaction library for Selenium.

We conclude that fingerprint hiding – in the sense that first-party bot detection can be mostly prevented – is effective. However, we find that spoofing properties in JavaScript can lead to website breakage. This observation, in general, is not new, but we were surprised that even simple changes to only the `navigator` object already caused breaks. As such, we advise investigating the compatibility of stealth plugins before using them in large-scale studies. HLISA is publicly available as a Python library [ART-IMC21]. Our spoofing extension can be found in the official OpenWPM repository [ART-IMC21-2].

To the best of our knowledge, HLISA is the first comprehensive interaction API that allows Selenium-based bots to hide identifiable behaviour. Based on the conceptual evaluation of an arms race between interaction detectors and simulators, HLISA significantly raises the bar for detectors. Before HLISA, bot interaction was detectable by its artificial nature. To detect HLISA, an interaction-based detector must compare the observed interaction to a model of human behaviour. We hope our findings help the research community to improve the reliability of web measurements.

Future work. HLISA's approximation of human behaviour can be further defined. First of all, HLISA's models of human interaction are based on an extremely small set of data. Extending the experiments with more subjects can improve the models underlying HLISA's interaction. Secondly, the interaction of scrolling and mouse movement can be further refined. Mouse movements especially are a rich source for analysis in studies in the human-computer interaction field [PT01; LB13; Fit54]. Related, the simulation of scrolling could be furthered to account for Firefox's `smooth scrolling` setting. With the availability of large data sets, HLISA could also leverage machine learning to generate interaction.

The conceptual discussion of HLISA's limitations offers a framework to reason about its capabilities but lacks concrete data. A practical evaluation would be desirable, but such necessitates detectors. To the best of our knowledge, no study to date has proposed a methodology to identify unknown interaction detectors in the wild with certainty.

Finally, our findings with respect to side effects of fingerprint overriding suggest that an investigation comparing stealth extensions is merited. In particular, such an investigation should elucidate how often they cause errors on the visited site and how often they are detected.

Chapter 9

Case Study: Overcoming specific Bot Detection

The premise of web measurement frameworks is that they measure what regular browsers would encounter on the Web. In practice, deviations due to the detection of automation have been found. To what extent automated browsers can be improved to reduce such deviations has not been investigated in detail so far. This chapter investigates a specific web automation framework: OpenWPM, a popular research framework specifically designed to study web privacy. We analyse (1) detectability of OpenWPM, (2) resilience of OpenWPM's data recording, and (3) prevalence of OpenWPM detection.

Our analysis (1) reveals that OpenWPM is easily detectable. Our investigation of OpenWPM's data recording integrity (2) identifies novel evasion techniques and previously unknown attacks against OpenWPM's instrumentation. We investigate and develop mitigations to address the identified issues. Finally, in a scan of 100,000 sites (3), we observe that OpenWPM is commonly detected (∼14% of front pages). Moreover, we discover integrated routines in scripts specifically to detect OpenWPM clients. In conclusion, our case study shows that even the most popular web measurement framework, OpenWPM, is more gullible than expected, and this gullibility is rarely accounted for in studies.

This chapter is based on the following publication:

How gullible are web measurement tools? A case study analysing and strengthening OpenWPM's reliability. Benjamin Krumnow, Hugo Jonker, and Stefan Karsch. *In Proc. 18th International Conference on emerging Networking EXperiments and Technologies (CoNEXT'22)*, DOI: 10.1145/3555050.3569131, 2022, [CoNEXT22].

9.1 Introduction

The goal of web studies is to analyse what regular visitors would experience on the Web. This relies on an (often unstated) assumption that the data as collected is representative of what human-controlled browser would encounter. Previous investigations [EN16; ESORICS19; ZBO+20; JSS+21] have shown that this is not always the case: websites have been found to omit content (advertisements, video, JavaScript execution, login forms, etc.) or require completion of a CAPTCHA for automated clients. This casts doubts on the validity of such analyses, especially since this effect is often not accounted for (see Section 9.2). In some cases, studies implement their own automation tooling [KL21; CLB+22] to bypass detectors targeting such frameworks. While such an approach can be technically sufficient, the additional steps may also shift or even exacerbate the detection problem (see Section 8.2). Another approach is to use third-party plugins that hide differences in the fingerprint (e.g., [JSS+21]). Again, this could be a reasonable approach, but 100% fidelity should not be assumed – though in practice, it often is. This raises the question: how gullible are frameworks as measurement tools? That is, to what extent can websites fool web measurement tools?

To analyse a website, a web measurement tool visits the site and collects whatever data it needs for its analysis. In our view, there are two key issues that affect whether websites can fool the tool. First: detectability. Detectability enables a website to deliver innocuous content to analysers instead of what regular visitors would receive (a so-called cloaking attack, see [ITK+16]). Second, the resilience of data collection, that is: able to collect sought-after data even in adverse conditions. Websites may employ obfuscation and other tricks to thwart analysis, e.g., sprinkling random breakpoints throughout the code to hinder analysis [MJ21]. A malicious website would use tricks specifically targeting analysers to hide its wrongdoings.

We investigate the extent to which these two issues affect the reliability of web measurement tools by means of a case study. Many web measurement tools are one-off creations, used to perform a specific analysis but not designed for use cases beyond that (e.g., [HSL+14; LSK+16; BAR+16; ESORICS19]). Amongst the few more general web measurement frameworks, most have not gained traction in the community and have been used little beyond their initial study so far (e.g., FPDetective [AJN+13], Crawlium [MHB+17], VisibleV8 [JK19]). In contrast, OpenWPM, a framework [EN16] for measuring web privacy, has been used in over 70 peer-reviewed publications (see Section 9.2). Studies based on OpenWPM keep frequently appearing in top conferences (see Table C.2). Its maturity, as well as its popularity and impact in the web measurement community, make it an ideal subject for our case study.

OpenWPM offers increased stability, fidelity and easy access to measurement functionality on top of Selenium + WebDriver (a browser automation framework). The framework can be run under either Ubuntu or macOS. It consists of four parts (Figure 9.1): a web client, automation components, instrumentation for measurements, and a framework. As a web client, OpenWPM uses an unbranded Firefox browser. In contrast to a regular Firefox browser, this allows running unsigned browser extensions. There are various measurement instruments implemented via one browser extension. Each facilitates recording a specific aspect, such as JavaScript calls, cookies, or HTTP traffic. Last is the framework, which acts as a maestro orchestrating

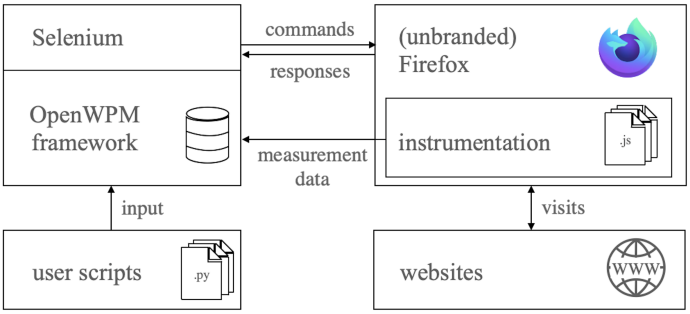

Figure 9.1: Components of the OpenWPM framework

work. Its purpose is to control browsers and data collection. It also adds various functionalities, such as monitoring for browser crashes and liveliness, restoring after failures, loading input data, etc.

Contributions. In this chapter, we provide the first thorough analysis of OpenWPM's reliability, which involves four main contributions.

- We provide the first analysis of OpenWPM's detectability based on both conventional fingerprinting (see Section 7.4) and template attacks [SLG19] techniques. We find previously not reported, identifiable properties for every mode of running OpenWPM (headless, Xvfb, etc.), even allowing to distinguish between these modes.
- We look for bot detectors in the Tranco Top 100K sites via both static and dynamic analysis. We find a drastic increase of Selenium-based bot detection compared to earlier studies. In addition, we find detectors in the wild specifically targeting OpenWPM.
- We explore how sites can attack OpenWPM's data collection. We find various attack vectors targeting OpenWPM's most commonly used instruments and implement proof-of-concept attacks for these.
- We harden OpenWPM against poisoning attacks and detection. This hides all identifiable properties when run in regular mode and addresses the identified attacks against OpenWPM's instrumentation. We evaluate its performance against vanilla OpenWPM. The number of cookies received is severely impacted. Conversely, ads/tracker traffic is hardly impacted.

Ethics and responsible disclosure. Our work aims to make OpenWPM a more reliable measurement framework. We responsibly disclosed our findings and shared fixes of the identified issues. This helps to make OpenWPM less detectable and, therefore, its results more reliable. Of course, a less detectable web bot may itself be abused. For attacking specific sites, our improvements do not greatly impact the attack surface: a less detectable OpenWPM is a fine tool for studying the Web, but not for a targeted attack on a specific site. For attacks that span thousands of sites

Table 9.1: Measurement characteristics in 72 peer-reviewed studies that are built upon OpenWPM

Category	Studies
Measures	
– HTTP traffic	56
– cookies	35
– JavaScript	22
– other	6
Run mode	
– unspecified	59
– virtualisation	16
– headless	7
– regular mode	3
– Docker	2
– Xvfb	2

Category	Studies
Interaction	
– no interaction	55
– clicking	11
– scrolling	8
– typing	5
Subpages	
– not visited	53
– visited	19
Bot detection	
– ignored	55
– discussed	17
◦ uses mitigation	8

(e.g., click farming), our improvements do not help: disguising as a regular browser is in- sufficient to overcome contemporary defences. For that, site-specific fingerprints are needed [TJM15]. Thus, existing re-identification-based countermeasures (e.g., rate limiting) are not impacted.

Availability. Our stealth extension and collected data set (see Section 9.6) are available online for follow-up research [ART-CoNEXT22]. The OpenWPM maintainers presently integrate our stealth extension into OpenWPM's code base.[1]

Outline. In the remainder of this chapter, we provide a review of previous studies that used OpenWPM for their experiments (Section 9.2). Then, we investigate OpenWPM's detectability (Section 9.3) and use the results to scan the Web for sites with capabilities to detect OpenWPM (Section 9.4). After that, we take the perspective of a hostile website and evaluate various attacks against OpenWPM's data recording facilities (Section 9.5). In the last part, we report on our implementation of an improved OpenWPM version and test it against bot detectors in the wild (Section 9.6). We close this chapter with a review of our findings and recommendations for measurement tools and studies (Section 9.7).

9.2 Use of OpenWPM in previous studies

To understand how OpenWPM is being used, we review the different studies performed to date with OpenWPM. In June 2022, 76 works, of which 57 peer-reviewed, were listed[2] as using OpenWPM. We further added 15 recent studies that had not yet been listed. For each study, we check the following: what is measured, whether subpages are visited, whether interaction is used, and what run mode is used. Table 9.1 summarises our findings. For a detailed view per study, we refer to Appendix C.

[1] `https://github.com/openwpm/OpenWPM/pull/1036`
[2] `https://webtap.princeton.edu/software/`

The *measures* category tallies how many studies used OpenWPM's various measurement instruments: HTTP traffic, cookies, and JavaScript. Each of these measures may be impacted individually due to bot detection. We tally how many studies measure HTTP traffic, cookies, and JavaScript, respectively. Interestingly, while most studies use OpenWPM to record HTTP traffic, a few (e.g., [LWP+17; EAW+19; SII+19; CNS20]) have used it as automation instead as a measurement tool. These are tallied under 'other' in Table 9.1. The other categories pertain to aspects that may impact detectability. In each case, it is currently unknown whether these play a role in bot detection. With respect to the *interaction* category, we note that no study mentioned implementing interaction mechanisms. Therefore, we assume all studies used OpenWPM's default interaction functionality.

With respect to the *run mode* category, note that not all studies provide information about this. Nevertheless, the used run mode may impact detectability [HBB+14] and thus should be considered. We, therefore, consider all currently supported modes:

a. *unspecified*: does not specify mode,

b. *regular*: uses a full Firefox browser,

c. *headless*: uses Firefox version without a GUI,

d. *Xvfb*: as regular, with visual output redirected to a buffer,

e. *Docker*: runs OpenWPM within a Docker container,

f. *External virtualisation*: uses virtual machines to run OpenWPM, possibly in cloud infrastructure.

The modes *a* to *c* refer to the browser, while the others relate to the environment. Consequently, OpenWPM allows picking only one run mode which is either regular, headless, or Xvfb-mode. Docker or external virtualisation can be used in conjunction with one of the modes defining the browser.

Lastly, we track whether the studies considered *bot detection* at all and, if so, whether they used OpenWPM's built-in anti-detection features. Aside from studies investigating bot detection directly, only very few consider fingerprinting [UDH+20] or cloaking [LLZ+19; CUT+21] as a potential risk for valid results.

9.3 Fingerprint surface of OpenWPM

We begin by addressing the research question *how can OpenWPM be distinguished from human-controlled web clients?* In general, a website operator looking to identify OpenWPM clients can either probe for identifiable properties (i.e., fingerprinting) or attempt to recognise OpenWPM's interaction. The latter is due to Selenium, whose interaction we studied in detail in Chapter 8. Those results fully carry over to OpenWPM. This leaves uncertainty about how OpenWPM's fingerprint distinguishes it from other clients and bots. In line with previous works, we call that part of a browser fingerprint that distinguishes a particular type of client from other types the *fingerprint surface* [TJM15]. Determining the fingerprint surface of an OpenWPM client requires a way to find its properties that deviate from properties and values of other clients. In Chapter 7 we showed that it suffices to consider differences within

Table 9.2: Summary of deviating properties of each OpenWPM setup (v.017.0) contrasted with OpenWPM's Firefox (v.90)

	macOS 10.15		**Ubuntu 18.04**			**Docker 19.03.6**
	RM	HM	RM	HM	Xvfb	RM
navigator.webdriver is true	✓	✓	✓	✓	✓	✓
screen dimension prop.	✓	✓	✓	✓	✓	✓
screen position prop.	✓	✓	✓	✓	✓	✓
font enumeration	–	–	–	–	–	✓
timezone is 0	–	–	–	–	–	✓
navigator.languages prop.	–	43	–	43	–	–
deviating WebGL prop.	–	2,037	–	2,061	18	27
With instrumentation:						
- through tampering	+253	+253	+252	+252	+252	+252
- added custom functions	+1	+1	+1	+1	+1	+1

RM: Regular mode; HM: Headless mode; Xvfb: X virtual frame buffer mode.

the client's 'browser family', that is, fingerprint differences with those clients who use the same rendering engine and JavaScript engine. By comparing the results for multiple clients of the same browser family, differences unique to each client are brought to light. In previous works, two approaches for browser fingerprinting were used: probing a specific list of properties (see Section 7.5), or using an automated approach to traverse the DOM [SLG19]. While there is overlap between the results of these methods, neither offers a complete superset of the other. We combine the results of both approaches to determine the fingerprint surface.

Limitations. Fingerprint-based bot detection requires identifiable properties of bots, such as deviations from regular user clients or inconsistencies. While we use state-of-the-art tooling to identify outstanding characteristics in OpenWPM at the HTTP layer and above, we cannot guarantee that all properties or methods are covered. Furthermore, our method compares the fingerprint of OpenWPM to the fingerprint of a regular (equivalent version) Firefox. Any differences cannot be due to the browser then. However, this does not guarantee that these differences are unique compared to other browsers. To reduce the likelihood of this, we validate the found fingerprint surface against several other web browsers (see Section 9.3.3).

9.3.1 How recognisable is OpenWPM?

We determine OpenWPM's fingerprint surface by comparing its client to a standalone version of the same Firefox browser. Any differences must originate in the hosting environment, the framework itself, the base implementation, the added automation, or measurement components. Note that it is already well-known that OpenWPM's underlying automation component, Selenium, is trivially recognisable by the `navigator.webdriver` property,[3] which OpenWPM does not address. We are

[3] `https://www.w3.org/TR/webdriver/#x4-interface`

looking for further distinguishing aspects. To account for possible effects of the various run modes of OpenWPM on the fingerprint surface, we determine variations for each setup on Ubuntu and macOS. Table 9.2 summarises the identifying properties found. In addition to ways to recognise OpenWPM's instrumentation, we also identify ways to recognise display-less scraping (headless or Xvfb mode), and virtualised run mode. Thus, every mode of running OpenWPM is identifiable as a web bot. We also checked combinations of modes, such as using Docker and headless together. However, we could not gain new insights from these setups; wherefore we do not discuss them further.

Recognisable via screen resolution, window position. We found two new identification measures that work against all modes of OpenWPM: screen resolution, set by OpenWPM, and window position, set by the browser automation framework. OpenWPM screen properties use standard values and cannot be changed (see Table 9.3). On macOS, all browser instances will use the same absolute coordinates; on Ubuntu, each window shifts by the same offset when using regular mode.

Table 9.3: Screen properties for various configurations

OS	mode	resolution	window	x	y	offset (x,y)
macOS	regular	2560 x 1440	1366 x 683	23	4	0, 0
	headless	1366 x 768	1366 x 683	4	4	0, 0
Ubuntu	regular	2560 x 1440	1366 x 683	80	35	8, 8
	headless	1366 x 768	1366 x 683	0	0	0, 0
	xvfb	1366 x 768	1366 x 683	0	0	0, 0
	docker	2560 x 1440	1366 x 683	0	0	0, 0

Suppressing output → more identifiable. Suppressing output to display (by using Xvfb, headless, or Docker) adds a significant number of differences. In headless mode, the lack of a WebGL implementation leads to thousands of missing properties. We also observe that this mode adds 43 new properties to the `navigator.language` object. Xvfb mode uses a regular Firefox browser, which contains WebGL functionality. Nevertheless, Xvfb mode causes 5 changed and 13 missing properties. Interestingly, both headless and Xvfb mode allow the detection of missing user elements by accessing the property *screen.availTop*. This describes the first y-coordinate that does not belong to the user interface.[4] In display-less modes, this is always zero, while regular browsers have larger values.

Virtualisation leaves identifiable traces. Using OpenWPM's Docker container causes the WebGL vendor property to contain the term `VMware, Inc.` (Table 9.4) – clear evidence for the use of virtualisation [VRR+20]. In addition, the Docker environment reduces the number of available JavaScript fonts to one (Bitstream Vera Sans Mono), nor does it provide information about the time zone.

[4] `https://developer.mozilla.org/en-US/docs/Web/API/Screen/availTop`

Table 9.4: Selected deviations, Ubuntu no-display modes

mode	WebGL vendor	avail{Top—Left}
RM	*AMD AMD TAHITI*	27, 72
HM	*Null*	0, 0
Xvfb	*Mesa/X.org llvmpipe (LLVM 12.0.0,...)*	0, 0
Docker	*VMware, Inc. llvmpipe (LLVM 10.0.0,...)*	27, 72

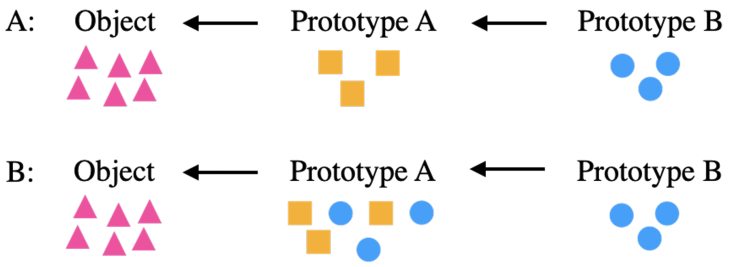

Figure 9.2: Properties in a (A) original object or (B) by the instrumentation polluted object

Using JS instrumentation has large effect on detectability. We checked if using any of OpenWPM's various instruments has any effect on its fingerprint surface. The only differences occur when using the JavaScript instrument. First, this instrument overwrites certain of the browser's standard JavaScript objects, which can be detected by using the `toString` function of a function or object (see Listing 9.1). Another identifying aspect of this instrument is the presence of a function in the window object (`window.getInstrumentJS`), which is not a part of the ECMA specification,[5] nor present in any common desktop browser (Firefox, Safari, Chrome, Edge, and Opera). Third, OpenWPM's wrapper functions can be found in stack traces. For that, a script needs to provoke an error in any overwritten function and catch the stack trace to successfully identify a modification by OpenWPM. Lastly, the JavaScript instrument 'pollutes' prototypes along the prototype chain of an object. Instrumenting is done by changing the prototype of an object, as well as all its ancestor prototypes. However, the properties of later ancestor prototypes are all added to the first ancestor prototype (see Figure 9.2). This distinguishes a visitor with instrumentation from one without.

9.3.2 How stable is the fingerprint surface?

We explored our determined fingerprint surface's stability, as new Firefox and OpenWPM versions may appear frequently. To that end, we repeated our experiments for older versions of OpenWPM (0.10.0 and 0.11.0). In general, we found that the fingerprint surfaces largely overlap. For example, on MacOS, the number of WebGL deviations in headless mode increases to 2,037 in OpenWPM 0.17.0, from OpenWPM 0.11.0's 2,022. In the oldest OpenWPM version (0.10.0), we find that the JavaScript

[5] `https://262.ecma-international.org/5.1/`

instrument adds two properties instead of one to the window object (`jsInstruments` and `instrumentFingerprintingApis`). In addition, we also investigated whether using an unbranded browser (as OpenWPM does) impacts OpenWPM's fingerprint. We did not find differences between branded and unbranded versions.

Using outdated browsers, however, does impact the fingerprint. For example, Google's reCAPTCHA service assigns a higher risk to older browser variants [SPK16c]. Releases of OpenWPM do not appear synchronously with Firefox. As a result, specific time frames exist where the OpenWPM client uses older Firefox versions than regular users. Table 9.5 summarises the migration of Firefox versions[6] in the OpenWPM Framework since version 0.10.0.[7] Between the release of Firefox 77 (March 2020) and Firefox 104 (current at the time of writing), were 780 days. The period where OpenWPM was shipped with an outdated version amounts to 540 days (69%).

Table 9.5: Migration to newer Firefox releases in OpenWPM

Firefox	release date	OpenWPM	integration date	outdated
104.0	07/23/22			53 days
101.0	05/31/22			
100.0	05/03/22	0.20.0	05/05/22	30 days
99.0	04/05/22			
98.0	03/08/22	0.19.0	03/10/22	58 days
96.0	01/11/22			
95.0	12/07/21	0.18.0	12/16/21	69 days
91.0	08/10/21			
90.0	07/13/21	0.17.0	07/24/21	11 days
89.0	06/01/21	0.16.0	06/10/21	9 days
88.0	04/19/21	0.15.0	05/10/21	48 days
87.0	03/23/21			
86.0.1	03/11/21	0.14.0	03/12/21	87 days
84.0	12/15/20			
83.0	11/18/20	0.13.0	11/19/20	58 days
81.0	09/22/20			
80.0	08/25/20	0.12.0	08/26/20	29 days
79.0	07/28/20			
78.0.1	07/01/20	0.11.0	07/09/20	9 days
78.0	06/30/20			
77.0	06/03/20	0.10.0	06/23/20	20 days

9.3.3 Validation of the fingerprint surface

Our measurement of OpenWPM's fingerprint surface should be validated to ensure our methodology did not introduce measurement artefacts. Moreover, our method only contrasts OpenWPM to Firefox; other browser fingerprints could contain elements from our measured fingerprint surface. To validate the fingerprint surface, we test its distinctiveness from other consumer browsers and platforms. We implemented an OpenWPM detector that uses four test strategies for the entire measured fingerprint surface to identify OpenWPM amongst web clients:

[6] These are listed on Mozilla's official site: `https://www.mozilla.org/en-US/firefox/releases/`

[7] We use dates as specified in OpenWPM's changelog: `https://github.com/openwpm/OpenWPM/commits/master/CHANGELOG.md`

```
window.canvas.getContext.toString();
// output of .toString when not instrumented
"function getContext() {
 [native code]
}"

// output of .toString when instrumented
"function () {
 const callContext = \
 getOriginatingScriptContext(!!logSettings.logCallStack);
 logCall(objectName + "." + \
 methodName, arguments, callContext, logSettings);
 return func.apply(this, arguments);
}"
```

Listing 9.1: Detectability of OpenWPM's JavaScript instrumentation

1. test for presence of a DOM property
2. test for absence of a DOM property
3. test if a native function was overwritten
4. compare a DOM property with an expected value

We tested the detector by setting up four machines, two MacBooks and two PCs, with Ubuntu. On each machine, we used OpenWPM and common browsers (Chrome, Safari, Opera and Firefox). We tested all distinguishing properties from Table 9.2. Our detector site was able to correctly identify OpenWPM every single time. Except for a few WebGL- and screen-related properties, all properties uniquely identify OpenWPM. As reported in Section 9.3.1, screen properties differ per operating system. For regular modes, the screen resolution depends on the system setup (display size and selected resolution). For WebGL properties, we found that these also occur on some non-OpenWPM clients (roughly 200 of 4K properties). After removing these, the fingerprint surface still contained a sizeable number of identifying WebGL and screen properties.

9.4 Incidence of OpenWPM detection

To assess the extent of OpenWPM detection in the wild, we conduct a large-scale measurement for client-side bot detection. In detail, we focus on scripts with capabilities to detect OpenWPM, i.e., scripts with routines to access properties unique for Selenium-based bots and OpenWPM. We find both general Selenium detectors and OpenWPM-specific detectors.

9.4.1 Data acquisition and classification

Methodology. Previous automated approaches to identify bot detectors have either relied on static (see Section 7.5) or dynamic analysis [JK19]. The idea behind static analysis is to identify code patterns in source code that link to known bot detectors

Table 9.6: Number of websites with Selenium detectors

# sites	static	dynamic	union
identified	32,694	19,139	38,264
without false positives / 'inconclusive'	15,838	16,762	18,714

or use specific bot-related properties. A limitation is that scripts may create code dynamically, which will be missed out by static analysis. Moreover, minification and obfuscation further increase the false negative rate of static analysis. The alternative approach, dynamic analysis, is to monitor JavaScript calls that identify a script as bot detector based on access to bot-related properties. Dynamic analysis does cover dynamically-generated scripts. Moreover, it does not monitor the code itself, only executed calls. An upside of this is that neither minification nor obfuscation affects the analysis. On the other hand, code that happens not to be executed during the run, is not analysed. Both static and dynamic analysis have been able to identify some bot detectors in the wild. It is unclear whether and to what extent the results of the methods differ in practice for finding web bot detectors. We combine both methods to increase coverage.

Setup. In order to assess the extent of client-side bot detection, we scan the top 100K websites of the Tranco list [LVT+19].[8] We set up an instance of OpenWPM running Firefox in regular mode. During a site visit, our OpenWPM client stores a copy of any transmitted JavaScript file and records JavaScript calls. We wait an additional 60 seconds after every completed page load[9] to give websites time to perform JavaScript operations. In addition, our client measures the presence of bot detection on subpages by opening a maximum of three URLs extracted from a site's landing page. For selecting subpages, we consider only URLs linking to the same domain. We use the scheme `eTLD+1` to identify domains. To account for websites that use same-origin requests to redirect clients to foreign domains, our client checks if a foreign domain was entered after following all redirects.

Accessing OpenWPM's fingerprint surface is cause to consider a script as a bot detector. However, certain scripts may access fingerprint surface attributes for other purposes, such as checking supported WebGL functionality. To reduce such false positives, we only classify a script as bot-detecting when it accesses properties pertaining to browser automation or are unique to OpenWPM (see Section 9.3.1). This leaves only the following: `navigator.webdriver`, which is specific to WebDriver-controlled bots; and the new identifying properties introduced by OpenWPM's instrumentation: `getInstrumentJS`, `instrumentFingerprintingApis`, and `jsInstruments`. Table 9.6 shows the results of the data collection and classification.

Limitations. Inherent in the above approach are several assumptions that can impact the results. First, our approach relies on the fingerprint surface we established. Detectors based on other methods (e.g., mouse tracking [CGK+13]) will be missed.

[8] `https://tranco-list.eu/list/WV79`
[9] As determined by the *document.readyState* property.

Second, we do not account for cross-site tracking. A third-party tracker could classify our client as a bot on one site and would need only to re-identify the client on another site, e.g., using IP filtering or regular browser fingerprinting. This amounts to a form of *website cloaking* – serving different content to specific clients. To what extent third-party tracking in general employs cloaking is a different study and left to future work. Both these limitations may cause an underestimation of the number of detectors (false negatives). As such, our approach approximates a lower bound on the number of detectors in the wild.

Pre-processing for static analysis. Within the static analysis, we pre-process scripts to undo straightforward obfuscation. We derive the respective encoding, transform hex literals to ASCII characters, and remove code comments. We apply our static analysis to scripts we collected during our scan of the Tranco Top 100K, resulting in 1,535,306 unique scripts.

To identify access to OpenWPM's fingerprint surface, we develop patterns. This process needs multiple iterations to reduce false positives. Our very first run used patterns matching strings literally. However, in the specific case of matching the term *webdriver*, we found that this selects scripts that use this word in another context than checking Selenium-driven Firefox browsers (cf., [JK19] and Chapter 7.5.2) for conflicting bot detection properties with this term). In the next iteration, we used patterns that take the context of the access to a property into account. For example, the pattern `navigator\[["']webdriver["']\]`only matches if the `webdriver` property is checked via the navigator object. Table 9.7 lists the patterns we explored in the scope of our study. Finally, we manually checked a random subset to check pattern performance. Only one pattern still introduced false positives; all its matches were manually validated and false positives eliminated.

Table 9.7: Patterns evaluated within the static analysis

pattern	false positives found
`webdriver`	✓
`instrumentFingerprintingApis`	-
`getInstrumentJS`	-
`jsInstruments`	-
`(?<!_|-)webdriver(?!_|-)`	✓
`navigator.webdriver`	-
`navigator\[["']webdriver["']\]`	-

Using honey properties to catch iterators. For the dynamic analysis, every recorded access to the fingerprint surface identifies a script with the potential to detect OpenWPM as a bot. This will also be triggered by scripts that iterate over all properties, e.g., for regular browser fingerprinting. Determining the purpose of such iteration requires per-script manual inspection and goes beyond dynamic analysis.

To determine whether property iteration occurs, we extend our client's navigator and window object with 'honey'[10] properties. These honey properties are added on

[10]We are not aware of any previous works using such an approach.

Table 9.8: Sites with scripts probing OpenWPM-specific properties

	cz	gs	google.com	ad1t
total	**331**	**14**	**9**	**2**
jsInstruments	331	5	2	2
instrumentFingerprintingApis	0	6	4	0
getInstrumentJS	0	3	3	0

cz: cheqzone.com, gs: googlesyndication.com, ad1t: adzouk1tag.com

the fly and use random strings as name. Hence, only a script using property iteration would access all honey properties. We divide scripts that use property iteration into two categories, based on access to the `navigator.webdriver` property: definitely detecting bots, and inconclusive. Iterator scripts are classified as inconclusive if they do not access `navigator.webdriver`, as all accesses to the fingerprint surface could be due to property iteration. Scripts that iterate the navigator object will naturally access the `webdriver` property. To check whether this access is only by iteration or intentional, we distinguish between scripts that trigger our static analysis and those that do not. Only scripts that do not surface in the static analysis are classified as inconclusive.

9.4.2 How often is OpenWPM detected?

OpenWPM can be detected directly via OpenWPM-specific properties or indirectly via properties of its underlying components (Selenium, WebDriver, etc.). Our results show that when checking both front- and subpages, at least 16.7% of the Tranco Top 100K websites execute scripts that accessed Selenium properties. Moreover, we also find four actors serving scripts that access OpenWPM-specific properties.

356 sites detect OpenWPM-specific properties. Most scripts we found recognise OpenWPM by targeting Selenium. A small number of detectors also include specific routines to detect OpenWPM itself. Overall, 356 sites executed scripts that accessed OpenWPM-specific properties. These scripts were all included via third-party domains belonging to four distinct providers. Table 9.8 summarises these detectors and their detection method. Detectors on cheqzone.com were found by static and dynamic analysis; detectors on the other three domains used some form of minification, obfuscation, or dynamic loading and were only found by dynamic analysis. We investigated the four hosting domains by consulting `whois` records, EasyList,[11] and the WhoTracksMe[12] database. All domains are related to the advertising industry. The domain `cheqzone.com` belongs to CHEQ, a company fighting ad fraud. The scripts hosted by Google domains are included through Google's reCAPTCHA service. While we could not clarify the origin of `adzouk1tag.com`, we found this domain listed in the EasyList for ad domains.

[11] `https://easylist.to/easylist/easylist.txt`
[12] `https://whotracks.me/`

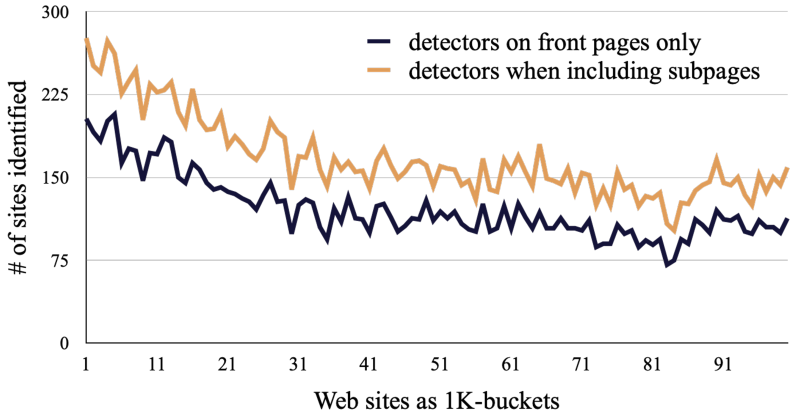

Figure 9.3: Sites with bot detectors on front- and subpages (depicted per 1K sites)

14% of sites detect bots on the front page. Figure 9.4 depicts the distribution of detectors active on the front page of websites for static and dynamic analysis. Dynamic analysis without considering property iteration identifies 12,208 sites with detectors on the front page. Static analysis measures the number of sites where bot detection could be triggered (11,897), including those where detection is present but not (yet) executed, e.g., where detection is only triggered after hovering over certain elements. While static and dynamic analysis identify a similar number of detectors for each bucket, they do not fully overlap. Combining both provides a slight increase in the presence of detectors (∼1.7K sites).

Deep scanning increases rate of detection by 5%-points. As discussed in Section 9.2, 26% of studies conducted with OpenWPM investigated subpages. This raises the question whether such studies are more often subject to bot detection, that is: does bot detection occur more frequently on subpages? Figure 9.3 depicts the occurrence of bot detectors on front pages and subpages. In general, studies examining subpages are at greater risk of being detected: the number of sites with active detectors increases by at least 37%. Hence, the average detection rate within the Top 100K sites will increase. That is: the study will be exposed to more detectors. Combining the results of both measurements, we see an increase of 5 percentage points (from 14% to 19%).

9.4.3 Who employs bot detection?

We separated detectors into first and third parties to check whether sites employ detection themself or use services by others. We find that the majority of sites include detectors from third-party domains. We count how often scripts on these third-party domains are included on scanned sites, tallying each third-party domain once per site. Some sites include more than one detector; hence the total number of inclusions exceeds the number of sites with detectors. Overall, we count 3,867 first-party detector scripts and 21,325 third-party detector scripts.

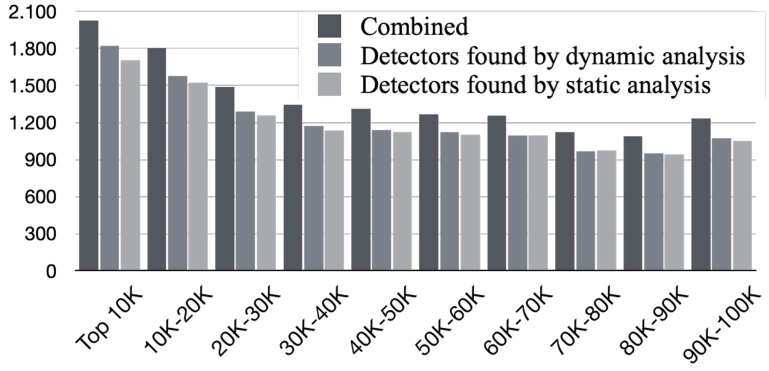

Figure 9.4: Detectors found on front pages

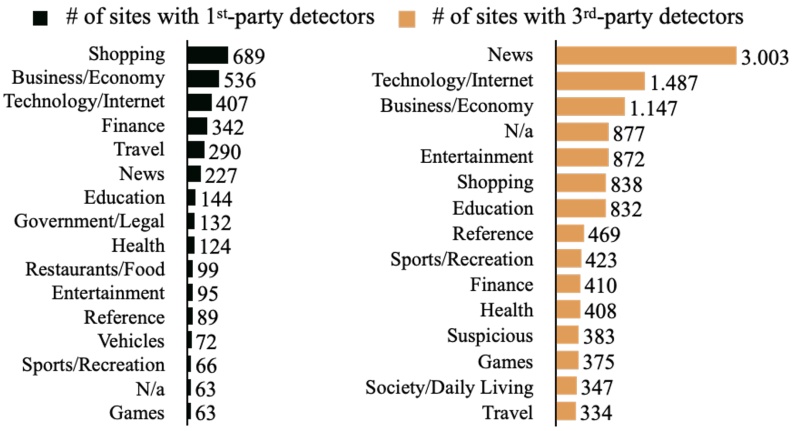

Figure 9.5: Common categories of sites with detectors

We explore what sites include detectors. We collect all categories for the identified 16K websites with detectors using Symantec's site review service (`https://sitereview.norton.com/`). Sites may be assigned multiple categories; we tally each listed category for such sites. Figure 9.5 depicts the 16 most often tallied categories for both first-party detectors (4,198 times) and third-party detectors (16,323 times). News sites are responsible for 18.4% of all third-party inclusions, followed by Technology (9%) and Business (7%). Interestingly, the ranks for Shopping (16.4%) and News (5%) switch for first-party detector inclusions. Moreover, sites in the categories Finance (8% vs 3%) and Travel (7% vs 2%) make up for a larger portion of the set of first-party inclusions than for third parties.

Third-party bot detection typically serves the advertisement industry. Following up on the previous point, we investigated the origins of third-party detectors. Table 9.9 breaks down the most commonly included domains. The top 10 domains account for two third of inclusions. WhoTracksMe categorises trackers ac-

Table 9.9: Domains hosting 3^{rd}-party detector scripts

	hosting domain	# inclusions (1/site)	%
	all	*21,325*	*100%*
1	yandex.ru	3,848	18.04%
2	adsafeprotected.com	2,309	10.83%
3	moatads.com	2,165	10.15%
4	webgains.io	2,091	9.81%
5	crazyegg.com	1,552	7.28%
6	intercomcdn.com	1,061	4.98%
7	teads.tv	854	4.00%
8	jsdelivr.net	423	1.98%
9	mxcdn.net	416	1.95%
10	mgid.com	402	1.89%
11+	*remaining 704 domains*	*6,204*	*29.1%*

cording to purpose. Using this, we find that the bot-detecting scripts on the most commonly included domains can serve a variety of purposes. For example, yandex.ru offers scripts used for advertising, content delivery network, site analytics, social media, and others. Other uses include web analytics (crazyegg.com), CDN (jsdelivr.net) and live chat (intercomcdn.com). However, bot detection is most commonly deployed by advertisers (e.g., domains 2,3,4,7,9, and 10 in Table 9.9).

The vast majority of first-party detectors are embedded third parties. To determine the origins of first-party bot detection scripts, we look for similarities between their inclusions of detectors. To do so, we hash the scripts and check for structural similarities in script URLs (see Table 9.10). Scripts provided by Akamai, Incapsula, Cloudflare, and PerimeterX follow the same script pattern and can be easily recognised. We found various similarities among unrelated sites. Scripts originating from Akamai occur the most frequently (1,004 sites). Second is Incapsula (998 sites), third is an unknown bot detector (659 sites), and fourth is Cloudflare (486 sites). Together, these top four originators account for 3,147 out of 3,867 sites (88%) where we found first-party detectors. In contrast to the purpose of third-party detectors, first-party detectors are not supplied by advertisement companies. Moreover, Akamai, Incapsula and Cloudflare all offer commercial bot detection services. With that in mind, one should expect sites with first-party detectors to tailor their responses to detected bots (e.g., throttling, blocking, withholding resources, and serving CAPTCHAs).

9.5 Attacking JavaScript recording

We have found detectors specifically targeting OpenWPM. This raises the question to what extent a malicious site could harm an OpenWPM study. We investigate whether a malicious website or third party could corrupt OpenWPM's data collection process. In particular, we consider an attacker that can deliver arbitrary content (HTML, cookies, JavaScript), but cannot break the browser's security model. To do so, our focus resides on attacks against the integrity or completeness of measurements. More specifically, we aim to attack the resilience of OpenWPM's most

Table 9.10: Similarities in first-party detectors

origin	URL path similarities	# sites
Akamai	domain/akam/11/...	1,004
Incapsula	domain/_Incapsula_Resource?...	998
Unknown	domain/asssets/{hash of 31-32 bits length} domain/resources/{hash of 32-33 bits length} domain/public/{hash of 32-33 bits length} domain/static/{hash of 34 bits length}	659
Cloudflare	domain/.../cdn-cgi/bm/cv/2172558837/api.js	486
PerimeterX	domain/.../{8 character string}/init.js	134

commonly used instruments: HTTP traffic, cookie recording, and JavaScript call recording. Both HTTP and cookie instruments are simple wrappers around browser functionality. Breaking them thus requires breaking the browser, which is outside the attacker model. The JavaScript instrument, on the other hand, needs to supply all its monitoring functionality itself. It is, therefore, clearly in the scope of our attacker model.

Since the instruments focus on data recording, we investigate attacks on data recording. More specifically, we consider:

1. whether data recording may be prevented;
2. whether fake data can be injected into the data recorder;
3. whether already recorded data can be deleted or altered;
4. finally, whether the data recording is complete.

Instruments in OpenWPM are implemented as a browser extension. Extensions are isolated to protect higher privilege APIs from access by untrusted code. Website scripts thus cannot directly interact with extensions. However, both extension and website scripts can read and change the DOM, opening the door for injection attacks against extensions that read the DOM. We conducted source code analysis for each instrument under investigation to identify vulnerabilities to such attacks. Below we discuss the found vulnerabilities.

Limitations. As we focus on data recording, the scope of our evaluation is limited to OpenWPM's instruments. Vulnerabilities could also be introduced by other OpenWPM components (see Section 9.1). Furthermore, we used manual code analysis; automated code analysis, such as code scanners or fuzzers, may give more results. To detect false positives, we validate the findings of the code analysis by implementing proof-of-concept attacks.

9.5.1 How to prevent recording?

We found two ways to prevent OpenWPM from recording JavaScript: disrupting communication to the data recorder and CSP stopping JavaScript injections.

By disrupting communication. We found a vulnerability that allows a website to turn off the recording of JavaScript calls in the JavaScript instrument. In more detail, the JavaScript instrument overwrites several API functions which use the event dispatcher to send messages when called. The event dispatcher then notifies the JavaScript instrument's back end to record the corresponding API call. To prevent an attacker from silently undoing these hooks, OpenWPM also hooks into (and thus: records access to) setters and getters to these API functions themselves. However, the event dispatcher itself is not protected. Thus, we can alter the event dispatcher to inject our own messages and manipulate messages sent to OpenWPM (Listing 9.2). To carry out this attack, the attacker overrides the event dispatcher to block all messages (all events from instrumented objects). This would already block OpenWPM recording by breaking any JavaScript API calls. However, this also would break a website's own JavaScript. To block only OpenWPM messages, the block needs to be tailored. Conveniently, tags messages with an ID to identify any monitored objects. Though this ID is randomly generated, it can easily be determined: simply trigger an API call to a monitored object, acquire the random ID from the observed message, and update the event dispatcher to only block messages containing this ID.

```
1  //Step I: Retrieve OpenWPM's random ID
2  function grabID() { return new Promise((resolve, reject) => {
3    let id;
4    document.dispatchEvent = function (event) {
5     id = event.type; document.dispatchEvent = dispatch_fn;
6     if (id !== undefined) { resolve(id);
7     } else { reject(new Error(msg));}
8    }
9    // Perform an action to grab the ID
10   navigator.userAgent;});}
11 // Step II: Overwrite event dispatcher to block events
12 async function attackExtension() {
13  let id = await grabID();
14  document.dispatchEvent = (event) => {
15   if (event.type != id) { dispatch_fn(event); // Dispatch event
16   } else {console.log("Event swallowed: " + event);}}}
```

Listing 9.2: Turn off the script recorder

By CSP stopping script injections. As stated above, OpenWPM overrides built-in browser functions. To do so, OpenWPM injects a script to the DOM to enter the page context, then calls that script to override the APIs, and finally, it removes the script from the DOM to hide what happened. This happens before the website's JavaScript code is loaded. However, websites may use Content Security Policy (CSP) to shield their users from XSS and data injection attacks. We found that the CSP `script-src` directive can be used to prevent OpenWPM's code injection. This will lead to CSP violation in the JavaScript instrument, leaving the API functions uninstrumented.

9.5.2 Can fake data be injected?

Altering the event dispatcher not only allows an attacker to block data recording, it also allows an attacker to learn the ID OpenWPM uses to record data. This is sufficient to inject almost arbitrary messages to be recorded. The attacker simply creates a custom event following the format used by OpenWPM's JavaScript extension and includes OpenWPM's assigned event ID. This enables an attacker to define most of the content of the resulting entry in OpenWPM's recording, such as the executing script URL or which function was called. Crucially, though, the website that originated the call is set outside of the browser by OpenWPM. The data sent by the event dispatcher is properly sanitised by the back-end, which prevents spoofing this. We can thus only inject fake data for the currently visited website. Note that a third party included on the site can also execute this attack.

9.5.3 Can records be deleted or altered?

Whereas the previous attacks exploited a vulnerability in the DOM-parsing front-end of the respective instruments, deleting already recorded data requires manipulating the instrument's back-end: SQLite. Attacking a database back-end requires an SQL injection vulnerability. We found that the current OpenWPM's data recording back-end (OpenWPM v0.20.0) properly sanitises its inputs; we deem this sufficient and did not investigate further.

9.5.4 Is data recording complete?

We evaluated whether data recording is complete or whether there are unobserved channels. We found two attacks against completeness: a bypass of the JavaScript instrument's recording and silent delivery of JavaScript code.

JS instrument's recording can be bypassed. We found a way to bypass OpenWPM's recording of JavaScript function calls. This attack again exploits OpenWPM's hooks to record function calls. In particular, the hooks must be attached to every object that is to be observed. For every new window or iframe, this must be done afresh. However, there is a long-standing bug in Chrome and Firefox [STK17], where both browsers, under some circumstances, fail to inject scripts into iframes. We tested if OpenWPM's implementation is affected by this and found that this is indeed the case.

Our evaluation of this attack involves two ways to access an iframe's DOM,[13] to create/execute iframes and their code: static vs. dynamic creation and immediate vs. delayed execution. Of these, immediate code execution (at creation time) is required to exploit this bug successfully. None of the other parameters we tested influenced the result. Listing 9.3 shows a proof-of-concept of this type of attack.

Silent delivery of JavaScript code. The HTTP instrument either stores all response bodies (full coverage) or it can be set to store JavaScript files only. The latter option significantly reduces stored content. The HTTP instrument should thus ensure

[13] `window.frames[0]` and `frame.contentWindow`

```
setTimeout(() => {
  let element = document.querySelector("#unobserved");
  let iframe = document.createElement('iframe');
  // HTML code for instantiating an iFrame
  iframe.src = "unobserved-iframe.html";
  element.appendChild(iframe);
  iframe.contentWindow.navigator.userAgent;
}, 500);
```

Listing 9.3: Example of an unobserved channel

recording of the aforementioned JavaScript attacks, unless this instrument's recording can also be bypassed. We managed to achieve this with an HTTP instrument only recording JavaScript files. For this mode, an attacker can silently deliver JavaScript code by sending it as text and, on the client side, convert it to code and execute it.

To successfully bypass OpenWPM's traffic recording of JS files, three aspects must be accounted for:

i. the content-type attribute must be set to something other than text/javascript;

ii. the src attribute must not contain a ".js" extension;

iii. the delivered file is not automatically executed; this must be handled by a different client-side script (e.g., using `eval()`).

Listing 9.4 shows an attack to bypass OpenWPM's mode of recording only JavaScript files. Note that the loaded resource does not include a file extension. Therefore, it will be loaded as regular text, and its content does not occur in OpenWPM's logging of loaded JavaScript files. After loading, the content is executed via *eval*.

```
const stealth_code = "https://{attacker_domain}/cheat";
fetch(stealth_code)               // load code from server
.then(res => res.text())      // convert code to JS-string
.then(res => eval(res));    // code execution
```

Listing 9.4: Example to silently load a JS file

9.6 Improving OpenWPM reliability

This section focuses on OpenWPM's reliability as an instrument measuring the Web as encountered by regular visitors. We explore how and to what extent reliability can be improved. To do so, we design an approach to hardening OpenWPM's instrumentation and hiding its distinctive fingerprint (from here on referred to as WPM_{hide}). Our proof-of-concept successfully hides the telltale signs of OpenWPM from its fingerprint and makes OpenWPM robust in the face of the discussed attacks in a lab setting. To evaluate its effectiveness in an open world setting, we run WPM_{hide} against detectors in the wild and contrast its measurements with those of a regular OpenWPM client.

9.6.1 How to hide the fingerprint?

OpenWPM's characteristic fingerprint varies with the various modes of running OpenWPM. For example, in headless Firefox mode, the fingerprint surface is difficult to hide due to headless mode's lack of functionality when compared to regular browsers. Hence, we focus on run modes where OpenWPM runs the browsers natively (Regular Mode). For such modes, we achieve stealth by overriding properties without leaving traces. These techniques can also be applied in other run modes (e.g., virtualisation).

The identifying properties for Regular Mode (see Table 9.2) relate to the `webdriver` property, window position, and dimension. Of OpenWPM's various instruments, only the JavaScript instrument causes additional identifiable properties. Hiding these properties can be achieved by customising the browser or including additional code inside a page's scope. Implementing the former requires significant work, but it can hide the fingerprint near-perfectly. The latter approach is far simpler to implement but risks leaving residual traces. We chose the second option for our proof-of-concept, as it can be seamlessly integrated within the current OpenWPM framework without significant effort.

Our proof-of-concept addresses all five identifiability issues (see Section 9.3.1):

(1) the `toString` operation of overwritten functions must return the regular (unchanged) output string;

(2) no additional property may appear in the DOM;

(3) stack traces must not show any signs of the instrumentation;

(4) prototype pollution must be avoided;

(5) prevent detection of automation components.

(1) Preserve `toString` output. For the first issue, we found that CanvasBlocker[14] addresses this well. Its implementation successfully fools all our fingerprinting tests (Section 9.3.1). CanvasBlocker creates a getter function with an identical signature to the function that must be overwritten and attaches it to the DOM based on a specific Firefox feature called `exportFunction`. The newly exported function is then used to redefine the getter of an object's prototype for a specific property. As a result, the overwritten function returns the native code string like a default browser property (Listing 9.1). Normally, accessing the getter of an object's prototype leads to an error. If this getter is replaced with a custom getter, that error is never thrown. This makes tampering with properties via an object's prototype detectable (see Section 8.2.1). Calling the original getter from the customised getter results in the original error being thrown, addressing this aspect of the fingerprint surface.

(2) Preserve clean DOM. The second issue arises during page load, prior to the page's JavaScript activation. The instrumentation injects its code as a script from the content context into the page context, overwrites the needed properties, and removes its code from the page context again. However, in practice, not all injected functions are deleted. We update the instrument to overwrite all functionality directly from the content context, thus keeping the page context clean.

[14] `https://github.com/kkapsner/CanvasBlocker`

(3) Fake stack traces. The third issue requires the stack trace to show no signs of instrumented functions. A web page can only access stack traces if errors occur. Usually, if an error occurs, the stack trace would show that the called function is called from inside the instrumentation. We address this by catching each error and throwing a new error with properly adjusted values for the file name, column, message, and line number.

(4) Avoid prototype pollution. The fourth issue relates to the pollution of an object's prototype. OpenWPM's instrument modifies only the first prototype in the prototype chain, not others further in the chain. We mitigate this by overwriting properties per prototype. Nevertheless, this mitigation has a limitation: it is not possible to determine the function's caller when a prototype has multiple descendants. This means our mitigation may inadvertently instrument more objects than intended. For prototypes located higher up the chain, the number of children increases, exacerbating this problem. Thankfully, most of the APIs that OpenWPM instruments by default are provided by prototypes close to the bottom of their prototype chain.

(5) Prevent detection of automation components. The automation components are detectable by window size, position, and webdriver attribute. For the latter, our hidden version must set the `navigator.webdriver` property to false like a regular Firefox browser. Since Firefox version 88, this flag is not user-settable.[15] We override the getter function of the `navigator.webdriver` property to return the default value (false) in the same fashion as described in the previous section. To change OpenWPM default window settings, we introduce a settings file that makes the window size and position settable in OpenWPM.

9.6.2 How to mitigate recording attacks?

Sending secure messages from page context to background context (see Section 9.5.1 and 9.5.2). As described in the previous section, a key benefit from migrating to Firefox's `exportFunction` is the ability to export higher privileged browser functions into the page. Hence, we can port functionality to the page context that is otherwise only available for content or background scripts of a browser extension. We use this to secure our instrumented functions, as we now can use the `browser.runtime` API to pass messages from the page to the background context. It is crucial that such functionality is exported to a private scope of an overwritten function to prevent access by other scripts in the page context. In addition, injecting functionality this way does not violate the CSP `script-src` directive. Thus, using the export function prevents the 'restriction of script injections', 'turn recording off' and 'inject fake data' attacks.

Intercepting DOM-modifying APIs (see Section 9.5.4). To address the tested variants of incomplete recordings, we use CanvasBlocker's frame protection. The basic idea is to intercept APIs used by page scripts to modify the DOM or create a new, non-instrumented copy of the DOM. This ensures that each modification or newly constructed DOM contains the instrumentation. Our implementation covers five cases:

[15] `https://bugzilla.mozilla.org/show_bug.cgi?id=1632821`

window constructors, DOM modification API, DOM creation via the `document.write` API, window mutations, and finally, the `window.open` API.

Filtered HTTP recording not robust (see Section 9.5.4). To the best of our knowledge, there is no known way to distinguish JavaScript code from text that is robust against a dedicated obfuscator. Therefore, an active adversary should be assumed to be capable of hiding JavaScript in a way that would accidentally be filtered out. Since this issue only arises in the presence of active adversaries, we recommend not using any filtering in such a case.

9.6.3 Does hardening impact measurements?

We developed a proof-of-concept implementation to hide the tell-tale signs of automation and to mitigate the found attacks. We evaluate the impact of our proof-of-concept implementation (from here on: WPM_{hide}) on web measurements when encountering bot detection in the wild. To that end, we contrast its results with vanilla OpenWPM (from here on: *WPM*) in HTTP traffic, cookies, JavaScript execution, and delivered JavaScript files. We test on all sites with bot detectors, as found by the analysis in Section 9.4. This list contains 1,487 sites with detectors. On these sites, we run *WPM* and WPM_{hide} in parallel (OpenWPM v.0.18.0, Firefox v.100, regular mode, HTTP, JavaScript, and cookie instrument activated) and configure each browser to idle 60 seconds on a page after loading completed. We use the latest version of Firefox on both machines; detection based on outdated browser versions thus does not apply to our evaluation (see Section 9.3.2). We take steps to mitigate noise in measurements. In particular, we avoid cross-client interference by separating both crawlers via two individual machines and IP addresses. These residential IP addresses are located in the same country, avoiding differences caused by geo-location and cloud-based IP blocking [ITK+16]. We synchronise visits between both machines to further reduce differences. Lastly, there is a risk that one-off events or single actors alter the measurements. To prevent the former, we repeat our measurement three times (r_1, r_2, and r_3). This allows us to check whether an effect persists or is only temporary. To address the latter, we test for significance. As the data sets are not normally distributed, we use the Wilcoxon signed-rank test with a confidence interval of 95%. In general, our findings show that *WPM* encounters less privacy-invasive behaviour than WPM_{hide}. We executed all three runs of our experiment one after another between the 20^{th} and 21^{st} of June 2022.

OpenWPM-induced CSP violations eliminated. Using WPM_{hide} results in a higher traffic volume, which increases with each run (see 'total' in Table 9.11). In order to find where this difference originates, we group requests by their resource type.[16] Table 9.11 breaks the differences down per resource type, showing results for data set r_1; proportions are similar for the other data sets. Most interesting are the changes in CSP reports. We see much fewer CSP reports for WPM_{hide}, as this version does not inject nodes into the DOM. This is also highly relevant, as CSP adoption and the directive for content restriction is on the rise [RBC+20]. We checked whether

[16] `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest/ResourceType`

Table 9.11: Comparison of HTTP request resource types

	r_1			r_2	r_3
Resource type	***WPM***	***WPM*$_{hide}$**	**diff.**	**diff.**	**diff.**
csp_report	784	188	-76.02%	-74.19%	-70.79%
media	530	610	+15.09%	-15.75%	-14.24%
beacon	5,951	6,622	+11.28%	+8.09%	+11.98%
websocket	321	302	-5.92%	-6.29%	-3.63%
xmlhttprequest	58,867	6,1702	+4.82%	+3.21%	+7.52%
imageset	5,730	5,982	+4.40%	+3.89%	+12.04%
font	9,608	9,356	-2.62%	-0.87%	-1.23%
object	50	49	-2.00%	0.00%	+6.38%
main_frame	3,955	3,883	-1.82%	-1.45%	-0.84%
image	116,296	118,068	+1.52%	+5.86%	+5.65%
script	83,239	84,385	+1.38%	+1.85%	+2.11%
sub_frame	15,393	15,592	+1.29%	+2.81%	+4.86%
other	95	96	+1.05%	-6.32%	+6.67%
stylesheet	9,943	10,028	+0.85%	+1.39%	+2.11%
Total	**310,737**	**316,673**	**+1.91%**	**+3.37%**	**+5.32%**

Table 9.12: HTTP requests to ad/tracker resources

	EasyList		EasyPrivacy	
	WPM	*WPM*$_{hide}$	*WPM*	*WPM*$_{hide}$
r_1	43,238	+1.64%	39,063	-1.64
r_2	41,659	+5.64%	37,710	+5.37
r_3	41,418	+5.81%	34,402	+7.85

any remaining CSP reports were due to *WPM*$_{hide}$, but none were. Note that the *WPM* column offers insights into how often *WPM* fails to install its hooks. In the worst case out of our three data sets, *WPM* failed to do so on 113 of 1,487 sites.

More ad/tracker HTTP-traffic. To assess the number of trackers and advertisers in traffic, we use the same approach as previous works [ADZ+20; JSS+21; CLB+22]: use the EasyList and EasyPrivacy blocklists[17] to identify trackers. Around a quarter of all HTTP traffic falls into this category. We further see that both machines encounter a significant difference in traffic by advertisers and trackers (p-value $<$ 0.0001). In most cases, this is a significant increase ($\sim$5%), though r_1 is an outlier in this regard (see Table 9.12).

Significantly more tracking cookies. For cookies, we contrasted the number of cookies between both variants per site. We find that the number of cookies served differs significantly for many sites, both for first parties as well as third parties (p-value $<$ 0.0001). As shown in Table 9.13, *WPM* receives fewer cookies, with the effect increasing each repetition (possibly due to *WPM* being re-identified). We see a similar effect in the number of sites that serve an unequal number of cookies to both machines: in r_1, 353 sites serve more cookies to *WPM*$_{hide}$ vs. 156 sites serving more

[17] `https://easylist.to/`

Table 9.13: Served cookies and differences with WPM_{hide}

	# first-party cookies		# third-party cookies		# tracking cookies	
	WPM	WPM_{hide}	*WPM*	WPM_{hide}	*WPM*	WPM_{hide}
r_1	28,826	+3.33%	31,335	+5.05%	3,031	+41.70%
r_2	28,841	+3.06%	30,977	+7.12%	2,929	+52.13%
r_3	28,744	+4.23%	29,692	+8.11%	2,719	+59.65%

cookies to *WPM*; this difference increases in r_3 to 394 sites for WPM_{hide} vs. 134 sites for *WPM*.

We investigated whether the difference in cookies was due to tracking cookies. To determine whether a cookie can be used for web tracking, we use the approach of Englehardt et al. [ERE+15], as refined by Chen et al. [CIP+21]. According to this method, a cookie may be used for tracking when:

1. it cannot be a session cookie,
2. the length of the cookie is eight or more characters (excluding surrounding quotes),
3. the cookie is always set,
4. the cookie is "long-living" (at least three months), and
5. the values differ significantly based on the Ratcliff-Obershelp algorithm [Bla21] among all runs.

In data set r_1, 3,031 cookies satisfy these criteria for *WPM*, while 4,295 cookies for WPM_{hide} match; a strong increase of 41.70%. This effect is again amplified in the other two runs.

Up to 37%-points more JS calls caught. As discussed in Section 9.5.4, *WPM*'s instrumentation does not cover all access methods. We track accessing API calls and their context in WPM_{hide} to determine what calls are recorded by WPM_{hide}, but missed by *WPM*. Figure 9.6 depicts the JavaScript APIs call in r_1 for WPM_{hide}. The black bar depicts the portion of calls covered by *WPM*; the rest is not. Some properties were almost entirely covered (Screen.top, 99% coverage), while others were not. Most prominently not covered is the Screen.availLeft API, where *WPM* records only 63% of calls that WPM_{hide} catches.

9.7 Conclusions

Our work calls into question the fidelity of web measurement tools. This fidelity has, so far, been mostly overlooked in measurement studies (see Table 9.1). We show that the most widely used web measurement tool, OpenWPM, is easily detectable by websites. We even found OpenWPM-specific detectors in practice. Moreover, this detectability may be leveraged by websites to hide actions from OpenWPM or even attack OpenWPM's functionality, undermining the fidelity of its results.

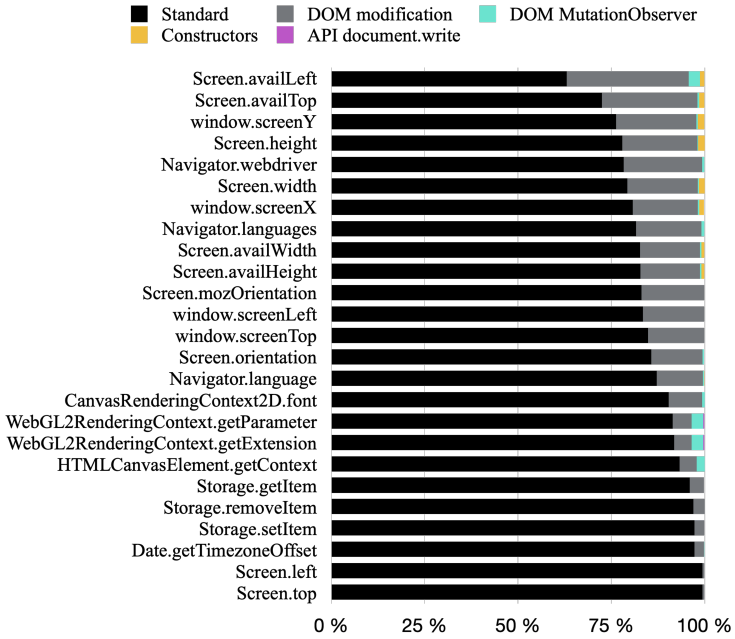

Figure 9.6: API calls in the context of DOM creation

This illustrates that web measurements should account for operating in a hostile environment. We show that OpenWPM can be hardened for such an environment, mitigating these adverse effects. However, similar caution should be taken with other web measurement tools. We have shown that the browser automation frameworks underpinning most measurement studies are themselves detectable. Our work should thus not be seen as an indictment of OpenWPM in favour of self-written one-off tooling. On the contrary, we need hardened tooling, which requires significant development effort. Our work is a call to action for web measurement studies to ensure any potential bot-induced bias in the measurement is eliminated. This means taking a hostile environment into account while developing tooling, and validating results specifically with a view to hostile actors.

Towards robust instrumentation. Our findings show that deployment of instruments via page context is fraught with difficulties. Ideally, the instrumentation is handled outside page scope, for example, by leveraging the debugger API. Unfortunately, Selenium v4 (the version used by OpenWPM) does not support this API currently. Alternatively, instrumentation could be integrated into the browser's source code. This supports great flexibility in hiding distinctive aspects of the browser fingerprint at the cost of significant additional maintenance overhead. This would slow the adoption of new browser versions; however, OpenWPM's adoption rate is already slow – the tradeoff may thus be worth it.

Selenium detection on the rise. In comparison with previous studies, we see that the number of sites looking for the `webdriver` property has significantly increased in

Table 9.14: Studies measuring `webdriver` property access on front pages

	when	**analysis**	**corpus**	**# sites**	**%**
[JK19]	2019–10	dynamic	Alexa 50K	2,756	5.51%
This work	2020–07	combined	Tranco 100K	13,989	13.99%
		– *static*		*11,957*	*11,96%*
		– *dynamic*		*12,194*	*12.19%*

the span of less than one year (see Table 9.14). This rapid change clearly suggests that websites are swiftly transitioning to responding differently to automated clients than to regular clients. Therefore, web studies should no longer ignore bot detection's potential impact on their study.

Recommendations for web measurement studies. In general, do not use virtualisation or headless or display-less modes. Studies that focus on measuring the amount of HTTP traffic seem to not be affected by detection and can, for now, get away with ignoring bot detection. In contrast, studies focussing on web tracking or cookies are affected (see Table 9.13) and must take bot detection into account. Finally, studies that automatically crawl beyond the front page will also be exposed to bot detectors significantly more often.

Recommendations for automated web measurement tooling. We identified two main challenges for measurement tooling: tooling resilience and reliability of its measurements. Tooling resilience requires assuming that the measured site is actively trying to break the measurement tool. This thus necessitates programming the measurement tooling defensively. Secondly, reliability of measurements is under pressure if the tooling's interactions with the measured objects deviate significantly from regular interactions. To minimise this effect, web measurement tools must take effort to blend in with human-originating traffic. Concretely, that includes avoiding DOM pollution as well as avoiding or reducing other tell-tale traces. Finally, these aspects should both be checked. That is: the detectability of the measurement tool should be checked using standard techniques, such as fingerprinting (cf., Chapter 7), template attacks [SLG19], and behaviour (cf., Chapter 8). Currently, there are no standardised ways to check measurement platforms for susceptibility to malicious data; however, our approach from Section 9.6.2 provides a starting point.

Part III

Final Remarks

Chapter 10

Conclusions

This thesis aims to overcome obstacles to automated web measurements and to determine the impact of doing so. To that end, it identifies two obstacles to automated measurements: those that treat automated visitors differently as a side effect of some other measure (unintended obstacles) and those that deliberately treat automated visitors differently (deliberate obstacles).

Overcoming unintended obstacles. In the first part of this thesis, we explore the automation of post-login studies and investigate differences emerging from view-based pricing. Our evaluation with Shepherd demonstrates that login studies can be scaled to thousands of sites while providing better efficacy than other login methods. At the same time, we adhere to ethical constraints by using crowd-sourced credentials, which, compared to the websites categories in the Tranco 1M list, do not show an untenable degree of bias. The latter is essential to consider Shepherd as a viable option for future post-login studies.

Further, our study of view-based price differentiation provides insights into the differences between desktop and mobile browsers, mobile apps, and localisation. For that, we design a framework that synchronises heterogeneous devices and use it to sample data simultaneously from multiple travel agencies. We conduct measurements on physical devices like desktops and mobiles to reduce differences from detection measures during the data sampling. Our study shows various indications for price differentiation among different views and underlines such comparisons' value. However, more work is needed to assess the extent of such practices.

Overcoming deliberate obstacles. In the second part, we investigate challenges in detecting bots through fingerprinting and behaviour to overcome deliberate obstacles. We address the challenge of finding detectable properties in bot frameworks by comparing bots to browsers with the same browser engine. We demonstrate that this is sufficient for identifying detectable properties for many popular automation frameworks. Moreover, our study of one million websites exhibits that detectors using such bot-identifying properties are widely prevalent. As many web studies rely on automation frameworks, our finding leaves a discomforting feeling about the reliability of web measurements.

With respect to behavioural detection, we encountered that Selenium's interaction is easily detectable by default. In addition, we found libraries for interaction simulation unsuited for measurement frameworks, as they need better integration and in-

creased API coverage. We address this with a Selenium-ready interaction API based on human-user interaction data. To classify our resulting library HLISA, we discussed a model that ranks detectors and simulators for interaction based on their capabilities. Our model and classification highlight that detectors should not rely on tests to determine whether interaction lies within human-possible boundaries. Instead, solid defences should check interaction for consistency to be prepared for frameworks like HLISA.

Assessing the effect of overcoming obstacles. We conducted one case study for each previously mentioned part to assess the impact of overcoming obstacles. The results of these case studies provide evidence that overcoming obstacles leads to more reliable and thorough results for web measurements.

By deploying Shepherd for a session security study, we observed that flaws widely spread through all stages of the session lifecycle. We even witnessed cases where hi-jacking threats endured a user's session termination as servers incorrectly upheld sessions. We conclude that the whole session lifecycle must be considered for thorough security evaluations of web sessions.

We further investigated the reliability of a popular web measurement framework. To that end, we tested possible vectors to attack and detect OpenWPM and used our findings to design a less detectable and hardened version. By conducting measurements with both versions, we could retrieve a before and after picture for the reliability of web measurements with this tool. Our results show that web studies without bot detection countermeasures risk a bias depending on what they measure.

10.1 Limitations

The research in this thesis provides a significant step in overcoming unintended obstacles. Our progress relies on an extensive engineering effort, especially for site-specific scrapers. This approach is acceptable for answering the research questions of this thesis, but these efforts are not readily transferable. In the future work section, we discuss several directions to address this.

Another limitation concerns the evaluation of anti-bot detection approaches. We evaluated our bot detection countermeasures to assess their effectiveness against commercial bot detectors we identified in the wild. Whereas looking for a bot fingerprint gives a concrete answer (yes or no), behavioural detection typically uses estimations and probabilities. This circumstance makes attributing changes in website responses to behavioural detection complex – which is further compounded by the fact that many websites use a combination of concrete and probabilistic techniques to identify bots. That is, it is likely but not 100% clear whether a change to a website was due to behavioural detection. How to test the efficacy of countermeasures against behavioural detection remains an open question.

Finally, overcoming obstacles requires the development of appropriate tooling, which can be a heavy burden for researchers. Hence, tools should be available to ease overcoming obstacles. The tools developed in this thesis work towards closing this gap. Nevertheless, propagating such tools can elevate automated attacks, which implies an ethical conflict. We encountered such a conflict during the sharing process

for Shepherd. On the one hand, shutting Shepherd away limits the potential for misuse. On the other hand, this strictly limits its value as a research tool. Therefore, we chose a middle ground by applying *responsible sharing*. In the next section, we discuss our approach and describe how to advance responsible sharing to become a standard procedure in the security community.

10.2 Discussion

May web studies ignore bot obstacles? Leaving obstacles to measurements unaddressed has been surprisingly common in some areas. We witnessed this during our literature review on session security studies (see Section 6.8). Similarly, only a small fraction of measurement studies account for bot detection (see Section 9.2 and the work by Demir et al. [DGU+22]). Given these insights, we wonder whether automated studies must account for obstacles. The answer to this question may hinge on two aspects: the prevalence of obstacles and their impact on a study's results. Concerning the prevalence, we see that impediments to automated measurements are common. We observed this for deliberate obstacles in two independent scans for bot detection (Chapters 7 and 9). Further, various features besides already highly present logins can result in unintentional obstacles to automation, e.g., responsive sites, GDPR cookie banners, geo-blocking and others.

Nevertheless, the community lacks a complete view of how obstacles impact results. Under this circumstance, we see three cases under which measurements should be judged: must account for obstacles, undecided or in-between, and can ignore them. This implies that the community should question the validity of recent studies falling into the first category when not accounting for obstacles. For older studies, however, conditions might divert from current conditions. Thus, repeating such studies may be required to evaluate whether a study's findings still hold.

How to responsibly share weaponisable research? Most tools developed for the studies in this thesis are available as open-source projects. For the Shepherd tool, we decided that the code is too easily weaponised to be made publicly available (see the discussion in Section 4.1). Nevertheless, we wanted to allow other researchers to build on our work. Therefore, we decided to share the tool responsibly. Thus, on the project page, we stated that we would make the code available under certain conditions. The conditions ensure adequate safeguards are in place to prevent misuse.[1]

Our case is certainly not exceptional, as security research often produces weaponisable tooling. Nevertheless, not making tooling available is a common and widely accepted practice. Missing access to tooling can negatively affect progress in research and may be a barrier to improvement. For example, the development of tooling can require – as sometimes demanded by reviewers – the comparison to competing solutions under equal conditions. Also, research tooling can be an important building block for developing advanced solutions. More importantly, access to tooling can contribute to discovering flaws, which will increase the robustness of research tooling in the long run. Therefore, we argue that this practice should be deprecated and re-

[1] The requirements for the Shepherd case are listed at `https://bkrumnow.github.io/shepherd/#availability`

placed with responsible sharing. This raises the question of how to set up responsible sharing. The obvious approach is to let the research group where the tool originated be responsible for sharing. However, that anoints them as gatekeepers, even though they have a conflict of interest. Moreover, their decisions may be biased, as they are most likely affected by what they share. Thus, responsible sharing should be a joint effort by the community rather than a single research group.

Practising responsible sharing for Shepherd over the last years, we could gather experiences that may be helpful to establish a fairer and broader applicable system. Overall, we see three steps that can remediate the problem above and improve the process:

1. Introducing mediators can address the gatekeeper situation. For that, responsible sharing could happen on an institutional or venue level, i.e., they evaluate requests, participate in defining conditions for sharing, and control access to tools. However, both solutions have advantages and disadvantages. For example, reviewers at scientific venues are already familiar with the research and are often experts in the field, which makes them suitable candidates for defining sharing conditions. In return, institutions are most likely aware of certain restrictions, such as local laws. Thus, a combination of both might be the best solution.

2. We see pressure on the institutional level as an important vehicle to bind a requester to their obligations, e.g., containing access to the source code, complying with given constraints, staying inside of ethical boundaries, and so forth. We implemented this for Shepherd by requiring the involvement of a permanent staff member who has authority and takes responsibility for a project.

3. Requirements for responsible sharing vary with a particular tool. Thus, conditions for sharing may be more permissive or restrictive. Expressing conditions accurately can be supported by providing a set of rules or specific licences. This is similar to the approach of open source software licences, but with the aim of restricting the proliferation of tools and the clarification of related aspects, such as necessary protection measures, handling of collected data, and so on.

10.3 Future work

The concepts and studies presented in this thesis open various avenues for future research. Given the results of our case studies, a promising direction is to measure the impact of overcoming obstacles for a wide range of privacy and security aspects. This includes deliberate and unintended obstacles, as well as combining both of them. To facilitate this, we must develop new tooling that inherently accounts for obstacles. Currently, available frameworks promise to satisfy either reliability or multi-view measurements. Future research should explore how to design instrumentation that is reliable but also portable to other platforms. This would also enable the application of the here presented concepts on a larger and more diverse set of clients, such as in-app web views [Ste18].

Another important direction for research is to control the influence of confounding factors (e.g., content updates, data replication among web servers, cross-site tracking,

etc.). While these can affect measurement results, there is no consensus on controlling them. Standardised controls would help the community to conduct more robust measurements. The first step toward this goal is the development of a taxonomy to gather potential confounding factors.

Finally, to make overcoming unintended obstacles more transferable, we envision a more generic approach to scraping, which would significantly reduce the engineering effort required to develop and maintain scrapers. A formal approach, e.g., based on a grammar for scraping, could prove a viable path for site-specific scrapers. For more generic scraping challenges, such as identifying specific actions (login/logout buttons), a multi-lingual machine-learning approach could provide a key step.

Bibliography

Author's Publications

[CoNEXT22] Benjamin Krumnow, Hugo Jonker, and Stefan Karsch. How gullible are web measurement tools? A case study analysing and strengthening OpenWPM's reliability. In *Proc. 18th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '22)*. ACM, 2022. ISBN: 9781450395083. DOI: 10.1145/3555050.3569131.

[CoSe21] Stefano Calzavara, Hugo Jonker, Benjamin Krumnow, and Alvise Rabitti. Measuring web session security at scale. *Comput. Secur.*, 111, 2021. DOI: 10.1016/j.cose.2021.102472.

[ESORICS19] Hugo Jonker, Benjamin Krumnow, and Gabry Vlot. Fingerprint surface-based detection of web bot detectors. In *Proc. 24th European Symposium on Research in Computer Security (ESORICS'19)*, volume 11736 of *LNCS*. Springer, 2019. DOI: 10.1007/978-3-030-29962-0_28.

[IMC21] Daniel Goßen, Hugo Jonker, Stefan Karsch, Benjamin Krumnow, and David Roefs. HLISA: towards a more reliable measurement tool. In *Proc. 21st ACM Internet Measurement Conference (IMC'21)*. ACM, 2021. DOI: 10.1145/3487552.3487843.

[MADWeb20] Hugo Jonker, Stefan Karsch, Benjamin Krumnow, and Marc Sleegers. Shepherd: a generic approach to automating website login. In *Proc. 2nd NDSS Workshop on Measurements, Attacks and Defenses for the Web (MADWeb'20)*. IEEE, 2020. DOI: 10.14722/madweb.2020.23008.

[MADWeb23] Hugo Jonker, Benjamin Krumnow, Godfried Meesters, and Stefan Karsch. Are some prices more equal than others? Evaluating store-based price differentiation. In *Proc. 5th NDSS Workshop on Measurements, Attacks and Defenses for the Web (MADWeb'23)*. IEEE, 2023. DOI: https://dx.doi.org/10.14722/madweb.2023.23011.

[SecWeb21] Luca Compagna, Hugo Jonker, Benjamin Krumnow, Johannes Krochewski, and Merve Sahin. A preliminary study on the adoption and effectiveness of SameSite cookies as a CSRF defence. In *Proc. 2nd (EuroS&P) Workshop on Designing Security for the Web (SecWeb'21)*. IEEE, 2021. DOI: 10.1109/EuroSPW54576.2021.00012.

Scholarly References and Printed Literature

[ABL+10] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. Towards a formal foundation of web security. In *Proc. 23rd IEEE Computer Security Foundations Symposium (CSF'10)*. IEEE Computer Society, 2010. DOI: 10.1109/CSF.2010.27.

[ACdS+07] Rui Alves, São Castro, Liliana de Sousa, and Sven Strömqvist. Influence of typing skill on pause-execution cycles in written composition. *Writing and Cognition: Research and Applications*, 2007. DOI: 10.1163/9781849508223_005.

[ADZ+20] Syed Suleman Ahmad, Muhammad Daniyal Dar, Muhammad Fareed Zaffar, Narseo Vallina-Rodriguez, and Rishab Nithyanand. Apophanies or epiphanies? How crawlers impact our understanding of the web. In *Proc. 29th The Web Conference (WWW'20)*. ACM, 2020. DOI: 10.1145/3366423.3380113.

[AEE+14] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juárez, Arvind Narayanan, and Claudia Díaz. The web never forgets: persistent tracking mechanisms in the wild. In *Proc. 21st ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*. ACM, 2014. DOI: 10.1145/2660267.2660347.

[AEN20] Gunes Acar, Steven Englehardt, and Arvind Narayanan. No boundaries: data exfiltration by third parties embedded on web pages. *Proc. Priv. Enhancing Technol.*, 2020(4), 2020. DOI: 10.2478/popets-2020-0070.

[AGH15] Ibrahim Altaweel, Nathan Good, and Chris Jay Hoofnagle. Web privacy census. *Technology Science*, 2015121502, 2015.

[AJ16] Amelia Andersdotter and Anders Jensen-Urstad. Evaluating websites and their adherence to data protection principles: tools and experiences - contributions to IFIP summer school proceedings. In *Privacy and Identity Management*, volume 498 of *IFIP Advances in Information and Communication Technology*, 2016. DOI: 10.1007/978-3-319-55783-0_4.

[AJN+13] Gunes Acar et al. Fpdetective: dusting the web for fingerprinters. In *Proc. 20th ACM Conference on Computer and Communications Security (CCS'13)*. ACM, 2013. DOI: 10.1145/2508859.2516674.

[AJP+20] Pushkal Agarwal, Sagar Joglekar, Panagiotis Papadopoulos, Nishanth Sastry, and Nicolas Kourtellis. Stop tracking me bro! Differential tracking of user demographics on hyperpartisan websites. In *Proc. 29th The Web Conference (WWW'20)*. ACM, 2020. DOI: 10.1145/3366423.3380221.

[AOM+19] Suzan Ali, Tousif Osman, Mohammad Mannan, and Amr M. Youssef. On privacy risks of public wifi captive portals. In *Data Privacy Management (DPM'19), Cryptocurrencies and Blockchain Technology (CBT'19)*, volume 11737 of *LNCS*. Springer, 2019. DOI: 10.1007/978-3-030-31500-9_6.

[ASL+20] Babak Amin Azad, Oleksii Starov, Pierre Laperdrix, and Nick Nikiforakis. Web runner 2049: evaluating third-party anti-bot services. In *Proc. 17th Detection of Intrusions and Malware, and Vulnerability Assessment DIMVA'20*, volume 12223 of *LNCS*. Springer, 2020. DOI: 10.1007/978-3-030-52683-2_7.

[BAA+15] Amy Brand, Liz Allen, Micah Altman, Marjorie M. K. Hlava, and Jo Scott. Beyond authorship: attribution, contribution, collaboration, and credit. *Learn. Publ.*, 28(2), 2015. DOI: 10.1087/20150211.

[BAR+16] Muhammad Ahmad Bashir, Sajjad Arshad, William K. Robertson, and Christo Wilson. Tracing information flows between ad exchanges using retargeted ads. In *Proc. 25th USENIX Security Symposium (USENIX Security'16)*. USENIX Association, 2016.

[BBF+10] Elie Bursztein, Steven Bethard, Celine Fabry, John C. Mitchell, and Daniel Jurafsky. How good are humans at solving captchas? A large scale evaluation. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010. DOI: 10.1109/SP.2010.31.

[BCF+14] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. Automatic and robust client-side protection for cookie-based sessions. In *Proc. 6th International Symposium on Engineering Secure Software and Systems (ESSoS'14)*. Springer, 2014. DOI: 10.1007/978-3-319-04897-0_11.

[BCF+15] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. CookiExt: patching the browser against session hijacking attacks. *J. Comput. Secur.*, 23(4), 2015. DOI: 10.3233/JCS-150529.

[BHR+12] Titus Barik, Brent E. Harrison, David L. Roberts, and Xuxian Jiang. Spatial game signatures for bot detection in social games. In *Proc. 8th Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-12)*. The AAAI Press, 2012.

[BKC+22] Dino Bollinger, Karel Kubicek, Carlos Cotrini, and David Basin. Automating cookie consent and GDPR violation detection. In *Proc. 31st USENIX Security Symposium (USENIX Security'22)*. USENIX Association, 2022.

[BLR+10] Douglas Brewer, Kang Li, Lakshmish Ramaswamy, and Calton Pu. A link obfuscation service to detect webbots. In *Proc. 6th IEEE International Conference on Services Computing (SCC'10)*, 2010. DOI: 10.1109/SCC.2010.89.

[BMP+16] Elie Bursztein, Artem Malyshev, Tadek Pietraszek, and Kurt Thomas. Picasso: lightweight device class fingerprinting for web clients. In *Proc. 6th Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'16)*. ACM, 2016.

[BRA+17] Justin Brookman, Phoebe Rouge, Aaron Alva, and Christina Yeung. Cross-device tracking: measurement and disclosures. *Proc. Priv. Enhancing Technol.*, 2017(2), 2017. DOI: 10.1515/popets-2017-0020.

[BSB+12] Elie Bursztein, Chinmay Soman, Dan Boneh, and John C. Mitchell. Sessionjuggler: secure web login from an untrusted terminal using session hijacking. In *Proc. 22nd International Conference on World Wide Web (WWW'12)*. ACM, 2012. DOI: 10.1145/2187836.2187880.

[BZK+18] Reuben Binns, Jun Zhao, Max Van Kleek, and Nigel Shadbolt. Measuring third-party tracker power across web and mobile. *ACM Trans. Internet Techn.*, 18(4), 2018. DOI: 10.1145/3176246.

[CCF+19] Stefano Calzavara, Mauro Conti, Riccardo Focardi, Alvise Rabitti, and Gabriele Tolomei. Mitch: A machine learning approach to the black-box detection of CSRF vulnerabilities. In *Proc. 4th IEEE European Symposium on Security and Privacy (EuroS&P'19)*. IEEE, 2019. DOI: 10.1109/EuroSP.2019.00045.

[CFN+19] Stefano Calzavara, Riccardo Focardi, Matús Nemec, Alvise Rabitti, and Marco Squarcina. Postcards from the post-HTTP world: amplification of HTTPS vulnerabilities in the web ecosystem. In *Proc. 40th IEEE Symposium on Security and Privacy (SP'19)*. IEEE Computer Society, 2019. DOI: 10.1109/SP.2019.00053.

[CFS+17] Stefano Calzavara, Riccardo Focardi, Marco Squarcina, and Mauro Tempesta. Surviving the web: A journey into web session security. *ACM Comput. Surv.*, 50(1), 2017. DOI: 10.1145/3038923.

[CGK+13] Zi Chu, Steven Gianvecchio, Aaron Koehl, Haining Wang, and Sushil Jajodia. Blog or block: detecting blog bots through behavioral biometrics. *Computer Networks*, 57(3), 2013.

[CHP+19] Vittoria Cozza, Van Tien Hoang, Marinella Petrocchi, and Rocco De Nicola. Transparency in keyword faceted search: an investigation on google shopping. In *Proc. 15th Italian Research Conference on Digital Libraries IRCDL'19*, volume 988 of *Communications in Computer and Information Science*. Springer, 2019. DOI: 10.1007/978-3-030-11226-4_3.

[CIP+21] Quan Chen, Panagiotis Ilia, Michalis Polychronakis, and Alexandros Kapravelos. Cookie swap party: abusing first-party cookies for web tracking. In *Proc. 30th The Web Conference (WWW'21)*. ACM / IW3C2, 2021.

[CLB+22] Darion Cassel et al. Omnicrawl: comprehensive measurement of web tracking with real desktop and mobile browsers. *Proc. Priv. Enhancing Technol.*, 2022(1), 2022. DOI: 10.2478/popets-2022-0012.

[CMC+15] Juan Miguel Carrascosa, Jakub Mikians, Ruben Cuevas, Vijay Erramilli, and Nikolaos Laoutaris. I always feel like somebody's watching me: *Measuring online behavioural advertising*. In *Proc. 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT'15)*, 2015. DOI: 10.1145/2716281.2836098.

[CMW15] Le Chen, Alan Mislove, and Christo Wilson. Peeking beneath the hood of uber. In *Proc. 15th ACM Internet Measurement Conference (IMC'15)*, 2015. DOI: 10.1145/2815675.2815681.

[CMW16] Le Chen, Alan Mislove, and Christo Wilson. An empirical analysis of algorithmic pricing on amazon marketplace. In *Proc. 25th International Conference on World Wide Web (WWW'16)*. ACM, 2016. DOI: 10.1145/2872427.2883089.

[CNH+13] Ping Chen, Nick Nikiforakis, Christophe Huygens, and Lieven Desmet. A dangerous mix: large-scale analysis of mixed-content websites. In *Proc. 16th Information Security Conference (ISC'13)*, volume 7807 of *LNCS*. Springer, 2013. DOI: 10.1007/978-3-319-27659-5_25.

[CNS20] John Cook, Rishab Nithyanand, and Zubair Shafiq. Inferring tracker-advertiser relationships in the online advertising ecosystem using header bidding. *Proc. Priv. Enhancing Technol.*, 2020(1), 2020. DOI: 10.2478/popets-2020-0005.

[CRB19] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Sub-session hijacking on the web: root causes and prevention. *Journal of Computer Security*, 27(2), 2019. DOI: 10.3233/JCS-181149.

[CTB+14] Stefano Calzavara, Gabriele Tolomei, Michele Bugliesi, and Salvatore Orlando. Quite a mess in my cookie jar! Leveraging machine learning to protect web authentication. *In Proc. 23rd international conference on World wide web (WWW'14)*, 2014. DOI: 10.1145/2566486.2568047.

[CTC+15] Stefano Calzavara, Gabriele Tolomei, Andrea Casini, Michele Bugliesi, and Salvatore Orlando. A supervised learning approach to protect client authentication on the web. *TWEB*, 9(3), 2015. DOI: 10.1145/2754933.

[CUT+21] Stefano Calzavara, Tobias Urban, Dennis Tatang, Marius Steffens, and Ben Stock. Reining in the web's inconsistencies with site policy. In *Proc. 28th Annual Network and Distributed System Security Symposium (NDSS'21)*. The Internet Society, 2021.

[CY19] Siebren Cosijn and Nataliya Yasko. *FP-Block 2.0: preventing browser fingerprinting*. Bachelor's thesis, Open University of the Netherlands, July 2019. https://www.open.ou.nl/hjo/supervision/2019-fpblock2-bsc-thesis.pdf.

[DAB+18] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. The web's sixth sense: A study of scripts accessing smartphone sensors. In *Proc. 25th ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*. ACM, 2018. DOI: 10.1145/3243734.3243860.

[DF20a] Ha Dao and Kensuke Fukuda. A machine learning approach for detecting CNAME cloaking-based tracking on the web. In *GLOBECOM*. IEEE, 2020.

[DF20b] Ha Dao and Kensuke Fukuda. Characterizing CNAME cloaking-based tracking on the web. In *Proc. 4th Network Traffic Measurement and Analysis Conference TMA'20*. IFIP, 2020.

[DG11] Derek Doran and Swapna S. Gokhale. Web robot detection techniques: overview and limitations. *Data Min. Knowl. Discov.*, 22(1-2), 2011.

[DGU+22] Nurullah Demir, Matteo Große-Kampmann, Tobias Urban, Christian Wressnegger, Thorsten Holz, and Norbert Pohlmann. Reproducibility and replicability of web measurement studies. In *Proc. 31st The Web Conference (WWW'22)*. ACM, 2022. DOI: 10.1145/3485447.3512214.

[DIP20] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. The cookie hunter: automated black-box auditing for web authentication and authorization flaws. In *Proc. 27th ACM SIGSAC Conference on Computer and Communications Security (CCS'20)*. ACM, 2020. ISBN: 9781450370899. DOI: 10.1145/3372297.3417869.

[DMF18] Ha Dao, Johan Mazel, and Kensuke Fukuda. Understanding abusive web resources: characteristics and counter-measures of malicious web resources and cryptocurrency mining. In *AINTEC*. ACM, 2018.

[DMF21] Ha Dao, Johan Mazel, and Kensuke Fukuda. CNAME cloaking-based tracking on the web: characterization, detection, and protection. *IEEE Trans. Netw. Serv. Manag.*, 18(3), 2021. DOI: 10.1109/TNSM.2021.3072874.

[dRND+12] Philippe de Ryck, Nick Nikiforakis, Lieven Desmet, Frank Piessens, and Wouter Joosen. Serene: self-reliant client-side protection against session fixation. In *Proc. IFIP International Conference on Distributed Applications and Interoperable Systems*, volume 7272 of *LNCS*. Springer, 2012. DOI: 10.1007/978-3-642-30823-9_5.

[EAW+19] Rob van Eijk, Hadi Asghari, Philipp Winter, and Arvind Narayanan. The impact of user location on cookie notices (inside and outside of the european union). In *Workshop on Technology and Consumer Protection (ConPro'19)*, 2019.

[Eck10] Peter Eckersley. How unique is your web browser? In *Proc. Priv. Enhancing Technol.* Volume 6205 of *LNCS*. Springer, 2010. DOI: 10.1007/978-3-642-14527-8_1.

[EHN18] Steven Englehardt, Jeffrey Han, and Arvind Narayanan. I never signed up for this! privacy implications of email tracking. *Proc. Priv. Enhancing Technol.*, 2018(1), 2018. DOI: 10.1515/popets-2018-0006.

[Eic95] David Eichmann. Ethical web agents. *Computer Networks and ISDN Systems*, 28(1), 1995. ISSN: 0169-7552. DOI: 10.1016/0169-7552(95)00107-3. Selected Papers from the Second World-Wide Web Conference.

[EN16] Steven Englehardt and Arvind Narayanan. Online tracking: a 1-million-site measurement and analysis. In *Proc. 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*. ACM, 2016. DOI: 10.1145/2976749.2978313.

[ERE+15] Steven Englehardt et al. Cookies that give you away: the surveillance implications of web tracking. In *Proc. 24th International Conference on World Wide Web (WWW'15)*. ACM, 2015. DOI: 10.1145/2736277.2741679.

[FBL+20] Imane Fouad, Nataliia Bielova, Arnaud Legout, and Natasa Sarafijanovic-Djukic. Missed by filter lists: detecting unknown third-party trackers with invisible pixels. *Proc. Priv. Enhancing Technol.*, 2020(2), 2020. DOI: 10.2478/popets-2020-0038.

[Fit54] Paul M Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology*, 47(6), 1954.

[FMS+15] Nathaniel Fruchter, Hsin Miao, Scott Stevenson, and Rebecca Balebako. Variations in tracking in relation to geographic location. *Proc. of the 9th Workshop on Web 2.0 Security and Privacy (W2SP) 2015*, 2015.

[FSK+20] Imane Fouad, Cristiana Santos, Feras Al Kassar, Nataliia Bielova, and Stefano Calzavara. On compliance of cookie purposes with the purpose specification principle. In *EuroS&P Workshops*. IEEE, 2020. DOI: 10.1109/EUROSPW51379.2020.00051.

[FSL+22] Imane Fouad, Cristiana Santos, Arnaud Legout, and Nataliia Bielova. My cookie is a phoenix: Detection, measurement, and lawfulness of cookie respawning with browser fingerprinting. *Proc. Priv. Enhancing Technol.*, 2022(3), 2022. DOI: 10.56553/popets-2022-0063.

[FZW15] Amin FaizKhademi, Mohammad Zulkernine, and Komminist Weldemariam. Fpguard: detection and prevention of browser fingerprinting. In *Proc. 29th Annual IFIP WG 11.3 Working Conference, DBSec*, volume 9149 of *LNCS*. Springer, 2015. DOI: 10.1007/978-3-319-20810-7_21.

[GG05] Alan Grosskurth and Michael W. Godfrey. A reference architecture for web browsers. In *Proc. 21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005. DOI: 10.1109/ICSM.2005.13.

[GGG+18] Chris Gibbs, Daniel Guttentag, Ulrike Gretzel, Lan Yao, and Jym Morton. Use of dynamic pricing strategies by airbnb hosts. *International Journal of Contemporary Hospitality Management*, 2018. DOI: 10.1108/IJCHM-09-2016-0540.

[GKR+18] Steven Goldfeder, Harry A. Kalodner, Dillon Reisman, and Arvind Narayanan. When the cookie meets the blockchain: privacy risks of web payments via cryptocurrencies. *Proc. Priv. Enhancing Technol.*, 2018(4), 2018. DOI: 10.1515/popets-2018-0038.

[GLL+14] Daniel Glez-Peña, Anália Lourenço, Hugo López-Fernández, Miguel Reboiro-Jato, and Florentino Fdez-Riverola. Web scraping technologies in an API world. *Briefings Bioinform.*, 15(5), 2014. DOI: 10.1093/bib/bbt026.

[GMC+14] Cristiano Giuffrida, Kamil Majdanik, Mauro Conti, and Herbert Bos. I sensed it was you: authenticating mobile users with sensor-enhanced keystroke dynamics. In *Proc. 11th Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'14)*, volume 8550 of *LNCS*. Springer, 2014. DOI: 10.1007/978-3-319-08509-8_6.

[Goß20] Daniel Goßen. *Design and implementation of a stealthy OpenWPM web scraper*. Bachelor's thesis, Open University of the Netherlands, April 2020. http://www.open.ou.nl/hjo/supervision/2020-d.gossen-bsc-thesis.pdf.

[GRC+18] Mohammad Ghasemisharif, Amrutha Ramesh, Stephen Checkoway, Chris Kanich, and Jason Polakis. O single sign-off, where art thou? an empirical analysis of single sign-on account hijacking and session management on the web. In *Proc. 27th USENIX Security Symposium (USENIX Security'18)*. USENIX Association, 2018.

[GVM+22] Meriem Guerar, Luca Verderame, Mauro Migliardi, Francesco Palmieri, and Alessio Merlo. Gotta CAPTCHA 'em all: A survey of 20 years of the human-or-computer dilemma. *ACM Comput. Surv.*, 54(9), 2022. DOI: 10.1145/3477142.

[GWX+09] Steven Gianvecchio, Zhenyu Wu, Mengjun Xie, and Haining Wang. Battle of botcraft: fighting bots in online games with human observational proofs. In *Proc. 16th ACM Conference on Computer and Communications Security (CCS'09)*. ACM, 2009. DOI: 10.1145/1653662.1653694.

[HA12] Shiva Houshmand and Sudhir Aggarwal. Building better passwords using probabilistic techniques. In *Proc. 28th Annual Computer Security Applications Conference (ACSAC'12)*. ACM, 2012. DOI: 10.1145/2420950.2420966.

[HBB+14] Grant Ho, Dan Boneh, Lucas Ballard, and Niels Provos. Tick tock: building browser red pills from timing side channels. In *WOOT*. USENIX Association, 2014.

[HdTS20] Xuehui Hu, Guillermo Suarez de Tangil, and Nishanth Sastry. Multi-country study of third party trackers from real browser histories. In *Proc. 6th IEEE European Symposium on Security and Privacy (EuroS&P'20)*. IEEE, 2020. DOI: 10.1109/EuroSP48549.2020.00013.

[HDU+21] Henry Hosseini, Martin Degeling, Christine Utz, and Thomas Hupperich. Unifying privacy policy detection. *Proc. Priv. Enhancing Technol.*, 2021(4), 2021. DOI: 10.2478/popets-2021-0081.

[HSL+14] Aniko Hannak, Gary Soeller, David Lazer, Alan Mislove, and Christo Wilson. Measuring price discrimination and steering on e-commerce web sites. In *Proc. 14th Internet Measurement Conference (IMC'14)*. ACM, 2014. DOI: 10.1145/2663716.2663744.

[HTW+18] Thomas Hupperich, Dennis Tatang, Nicolai Wilkop, and Thorsten Holz. An empirical study on online price differentiation. In *Proc. 8th ACM Conference on Data and Application Security and Privacy (CODASPY'18)*. ACM, 2018. DOI: 10.1145/3176258.3176338.

[IES21] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. Fingerprinting the fingerprinters: learning to detect browser fingerprinting behaviors. In *Proc. 42nd IEEE Symposium on Security and Privacy (S&P'21)*, 2021. DOI: 10.1109/SP40001.2021.00017.

[ISS+17] Costas Iordanou, Claudio Soriente, Michael Sirivianos, and Nikolaos Laoutaris. Who is fiddling with prices?: building and deploying a watchdog service for e-commerce. In *Proc. 31st Conference of the ACM Special Interest Group on Data Communication, SIGCOMM'17*, 2017. DOI: 10.1145/3098822.3098850.

[ITK+16] Luca Invernizzi, Kurt Thomas, Alexandros Kapravelos, Oxana Comanescu, Jean Michel Picod, and Elie Bursztein. Cloak of visibility: detecting when machines browse a different web. In *Proc. 37th IEEE Symposium on Security and Privacy (S&P'16)*, 2016. DOI: 10.1109/SP.2016.50.

[IWN+22] Umar Iqbal, Charlie Wolfe, Charles Nguyen, Steven Englehardt, and Zubair Shafiq. Khaleesi: breaker of advertising and tracking request chains. In *Proc. 31st USENIX Security Symposium (USENIX Security'22)*. USENIX Association, 2022. ISBN: 978-1-939133-31-1.

[JBS+11] Martin Johns, Bastian Braun, Michael Schrank, and Joachim Posegga. Reliable protection against session fixation attacks. In *Proc. 26th ACM Symposium on Applied Computing (SAC'16)*. ACM, 2011. DOI: 10.1145/1982185.1982511.

[JK19] Jordan Jueckstock and Alexandros Kapravelos. Visiblev8: in-browser monitoring of javascript in the wild. In *Proc. 19th ACM Internet Measurement Conference (IMC'19)*. ACM, 2019. DOI: 10.1145/3355369.3355599.

[JSS+21] Jordan Jueckstock et al. Towards realistic and reproducible web crawl measurements. In *Proc. 30th The Web Conference (WWW'21)*. ACM, 2021. DOI: 10.1145/3442381.3450050.

[Kau17] Lutz Kaulfuß. Flugbuchung per Smartphone Achtung! Neue Abzocke! *Clever Reisen!*, (1), 2017.

[KB15] Michael Kranch and Joseph Bonneau. HTTPS in mid-air: an empirical study of strict transport security and key pinning. In *Proc. 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*. The Internet Society, 2015.

[KIS+20] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. Carnus: exploring the privacy threats of browser extension fingerprinting. In *Proc. 27th Annual Network and Distributed System Security Symposium (NDSS'20)*. The Internet Society, 2020.

[KL21] Dhruv Kuchhal and Frank Li. Knock and talk: investigating local network communications on websites. In *Proc. 21st ACM Internet Measurement Conference (IMC'21)*. ACM, 2021. DOI: 10.1145/3487552.3487857.

[KTK20] Martin Koop, Erik Tews, and Stefan Katzenbeisser. In-depth evaluation of redirect tracking and link usage. *Proc. Priv. Enhancing Technol.*, 2020(4), 2020. DOI: 10.2478/popets-2020-0079.

[LB13] Byungjoo Lee and Hyunwoo Bang. A kinematic analysis of directional effects on mouse control. *Journal of Ergonomics*, 56, 2013. DOI: 10.1080/00140139.2013.835074.

[LBB+20] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. Browser fingerprinting: a survey. *ACM Transactions on the Web (TWEB)*, 14(2), 2020.

[LBM17] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. Fprandom: randomizing core browser objects to break advanced device fingerprinting techniques. In *Proc. 9th Symposium of the Engineering Secure Software and Systems (ESSoS'17)*, volume 10379 of *LNCS*. Springer, 2017. DOI: 10.1007/978-3-319-62105-0_7.

[LLZ+19] Baojun Liu et al. Traffickstop: detecting and measuring illicit traffic monetization through large-scale DNS analysis. In *Proc. 4th IEEE European Symposium on Security and Privacy (EuroS&P'19)*. IEEE, 2019. DOI: 10.1109/EuroSP.2019.00047.

[LSK+16] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: an archaeological study of web tracking from 1996 to 2016. In *Proc. 25th USENIX Security Symposium (USENIX Security'16)*. USENIX Association, 2016.

[LVT+19] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: a research-oriented top sites ranking hardened against manipulation. In *Proc. 26th Annual Network and Distributed System Security Symposium (NDSS'19)*. The Internet Society, 2019. DOI: 10.14722/ndss.2019.23386.

[LWP+17] Fang Liu, Chun Wang, Andres Pico, Danfeng Yao, and Gang Wang. Measuring the insecurity of mobile deep links of android. In *Proc. 26th USENIX Security Symposium (USENIX Security'17)*. USENIX Association, 2017.

[MAF+19] Arunesh Mathur et al. Dark patterns at scale: findings from a crawl of 11k shopping websites. *Proc. ACM Hum. Comput. Interact.*, 3(CSCW), 2019. DOI: 10.1145/3359183.

[Mee21] Godfried Meesters. *Synchronising Distributed Scraping*. Master's thesis, Open University of the Netherlands, August 2021. http://www.open.ou.nl/hjo/supervision/2021-godfried-meesters-msc-thesis.pdf.

[MFK16] Yogesh Mundada, Nick Feamster, and Balachander Krishnamurthy. Half-Baked Cookies: hardening cookie-based authentication for the modern web. In *Proc. 11th ACM Asia Conference on Computer and Communications Security (ASIACCS'16)*, 2016. DOI: 10.1145/2897845.2897889.

[MGE+12] Jakub Mikians, László Gyarmati, Vijay Erramilli, and Nikolaos Laoutaris. Detecting price and search discrimination on the internet. In *11th ACM Workshop on Hot Topics in Networks, HotNets-XI, Redmond, WA, USA - October 29 - 30, 2012*, 2012. DOI: 10.1145/2390231.2390245.

[MGE+13] Jakub Mikians, László Gyarmati, Vijay Erramilli, and Nikolaos Laoutaris. Crowd-assisted search for price discrimination in e-commerce: first results. In *Proc. 9th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT'13)*. ACM, 2013. DOI: 10.1145/2535372.2535415.

[MGF19] Johan Mazel, Richard Garnier, and Kensuke Fukuda. A comparison of web privacy protection techniques. *Comput. Commun.*, 144, 2019. DOI: 10.1016/j.comcom.2019.04.005.

[MHB+17] Georg Merzdovnik et al. Block me if you can: A large-scale study of tracker-blocking tools. In *Proc. 2nd IEEE European Symposium on Security and Privacy (EuroS&P'17)*. IEEE, 2017. DOI: 10.1109/EuroSP.2017.26.

[MJ21] Marius Musch and Martin Johns. U can't debug this: detecting javascript anti-debugging techniques in the wild. In *Proc. 30th USENIX Security Symposium (USENIX Security'21)*. USENIX Association, 2021.

[MLK+10] Marti Motoyama, Kirill Levchenko, Chris Kanich, Damon McCoy, Geoffrey M. Voelker, and Stefan Savage. Re: captchas-understanding captcha-solving services in an economic context. In *Proc. 19th USENIX Security Symposium (USENIX Security'10)*, 2010.

[MN22] Maaz Bin Musa and Rishab Nithyanand. ATOM: ad-network tomography. *Proc. Priv. Enhancing Technol.*, 2022(4), 2022. DOI: 10.56553/popets-2022-0110.

[MS12] K. Mowery and H. Shacham. Pixel Perfect: Fingerprinting canvas in HTML5. In *Proc. Web 2.0 Security & Privacy (W2SP '12)*. IEEE Computer Society, 2012.

[MSH19] Max Maass, Stephan Schwär, and Matthias Hollick. Towards transparency in email tracking. In *Proc. 7th Annual Privacy Forum (APF'19)*, volume 11498 of *LNCS*. Springer, 2019.

[MSN17] Najmeh Miramirkhani, Oleksii Starov, and Nick Nikiforakis. Dial one for scam: A large-scale analysis of technical support scams. In *Proc. 24th Annual Network and Distributed System Security Symposium (NDSS'17)*. The Internet Society, 2017.

[MWP+17] Max Maaß, Pascal Wichmann, Henning Pridöhl, and Dominik Herrmann. Privacyscore: improving privacy and security via crowd-sourced benchmarks of websites. In *APF*, volume 10518 of *LNCS*. Springer, 2017.

[NIK+12] Nick Nikiforakis et al. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proc. 19th ACM Conference on Computer and Communications Security (CCS'12)*, 2012. DOI: 10.1145/2382196.2382274.

[NJL15] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. PriVaricator: deceiving fingerprinters with little white lies. In *Proc. 24th International Conference on World Wide Web (WWW'15)*. ACM, 2015. DOI: 10.1145/2736277.2741090.

[NKJ+13] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: exploring the ecosystem of web-based device fingerprinting. In *Proc. 34th IEEE Symposium on Security and Privacy (SP'13)*. IEEE Computer Society, 2013. DOI: 10.1109/SP.2013.43.

[NMY+11] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. SessionShield: lightweight protection against session hijacking. In *Proc. 3rd Symposium on Engineering Secure Software and Systems (ESSoS'11)*, volume 6542 of *LNCS*. Springer, 2011. DOI: 10.1007/978-3-642-19125-1_7.

[Noo19] Robbert Noordzij. *Synthetic Fragmentation Experiments using WildFragSim*. Bachelor's thesis, Open University of the Netherlands, July 2019. https://www.open.ou.nl/hjo/supervision/2019-robbert.noordzij-bsc-thesis.pdf.

[OEN17] Lukasz Olejnik, Steven Englehardt, and Arvind Narayanan. Battery status not included: assessing privacy in web standards. In *Proc. 3rd International Workshop on Privacy Engineering (IWPE'17)*, volume 1873 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017.

[OSD+19] Adam Oest, Yeganeh Safaei, Adam Doupé, Gail-Joon Ahn, Brad Wardman, and Kevin Tyers. Phishfarm: A scalable framework for measuring the effectiveness of evasion techniques against browser phishing blacklists. In *Proc. 40th IEEE Symposium on Security and Privacy (S&P'19)*. IEEE, 2019. DOI: 10.1109/SP.2019.00049.

[OZW+20] Adam Oest et al. Sunrise to sunset: analyzing the end-to-end life cycle and effectiveness of phishing attacks at scale. In *Proc. 29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, 2020.

[PDA+20] Shahrooz Pouryousef, Muhammad Daniyal Dar, Suleman Ahmad, Phillipa Gill, and Rishab Nithyanand. Extortion or expansion? an investigation into the costs and consequences of ICANN's gTLD experiments. In *Proc. 21st Passive and Active Measurement (PAM'20)*, volume 12048 of *LNCS*. Springer, 2020. DOI: 10.1007/978-3-030-44081-7_9.

[PPL+06] KyoungSoo Park, Vivek S. Pai, Kang-Won Lee, and Seraphin B. Calo. Securing web service by automatic robot detection. In *Proc. USENIX Annual Technical Conference USENIX ATC'06*, 2006.

[PSF16] Kien Pham, Aécio S. R. Santos, and Juliana Freire. Understanding website behavior based on user agent. In *Proc. 39th International ACM conference on Research and Development in Information Retrieval SIGIR'16*, 2016. DOI: 10.1145/2911451.2914757.

[PT01] James Phillips and Thomas Triggs. Characteristics of cursor trajectories controlled by the computer mouse. *Journal of Ergonomics*, 44, 2001. DOI: 10.1080/00140130121560.

[RB14] Nicky Robinson and Joseph Bonneau. Cognitive disconnect: understanding facebook connect login permissions. In *Proc. 2nd ACM Conference on Online social networks (COSN'14)*. ACM, 2014. DOI: 10.1145/2660460.2660471.

[RBC+20] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. Complex security policy? A longitudinal analysis of deployed content security policies. In *Proc. 27th Annual Network and Distributed System Security Symposium (NDSS'20)*. The Internet Society, 2020.

[RCW+22] Sebastian Roth, Stefano Calzavara, Moritz Wilhelm, Alvise Rabitti, and Ben Stock. The security lottery: measuring client-side web security inconsistencies. In *Proc. 31st USENIX Security Symposium (USENIX Security'22)*. USENIX Association, 2022.

[RK17] Andrew Reed and Michael J. Kranch. Identifying https-protected netflix videos in real-time. In *Proc. 7th ACM Conference on Data and Application Security and Privacy (CODASPY'17)*. ACM, 2017.

[RKW12] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and defending against third-party tracking on the web. In *Proc. 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, 2012.

[RM21] Nathan Reitinger and Michelle L. Mazurek. ML-CB: machine learning canvas block. *Proc. Priv. Enhancing Technol.*, 2021(3), 2021. DOI: 10.2478/popets-2021-0056.

[Roe21] David Roefs. Camouflaging OpenWPM. Technical report, Open University of the Netherlands, June 2021. http://www.open.ou.nl/hjo/supervision/2021-david.roefs-research-internship.pdf.

[RSJ20] Nataasha Raul, Radha Shankarmani, and Padmaja Joshi. A comprehensive review of keystroke dynamics-based authentication mechanism. In *Proc. 2nd International Conference on Innovative Computing and Communications (ICICC'19)*. Springer, 2020. ISBN: 978-981-15-0324-5. DOI: 10.1007/978-981-15-0324-5_13.

[RTM21] Valentino Rizzo, Stefano Traverso, and Marco Mellia. Unveiling web fingerprinting in the wild via code mining and machine learning. *Proc. Priv. Enhancing Technol.*, 2021(1), 2021. DOI: 10.2478/popets-2021-0004.

[RUW19] Tarun Ramadorai, Antoine Uettwiller, and Ansgar Walther. The market for data privacy. Technical report, 2019. DOI: http://dx.doi.org/10.2139/ssrn.3352175.

[SAH+22] Asuman Senol, Gunes Acar, Mathias Humbert, and Frederik J. Zuiderveen Borgesius. Leaky forms: A study of email and password exfiltration before form submission. In *31st USENIX Security Symposium (USENIX'22)*. USENIX Association, 2022.

[SAM+22] Nayanamana Samarasinghe, Aashish Adhikari, Mohammad Mannan, and Amr M. Youssef. Et tu, brute? privacy analysis of government websites and mobile apps. In *Proc. 31st The Web Conference (WWW'22)*. ACM, 2022. DOI: 10.1145/3485447.3512223.

[SBC+15] Richard Shay et al. A spoonful of sugar?: the impact of guidance and feedback on password-creation behavior. In *Proc. 33rd ACM Conference on Human Factors in Computing Systems (CHI'15)*. ACM, 2015. DOI: 10.1145/2702123.2702586.

[Sch17] Steven Schmeiser. Online advertising networks and consumer perceptions of privacy. *Applied Economics Letters*, 25(11), 2017.

[SCL+21] Ido Sivan-Sevilla, Wenyi Chu, Xiaoyu Liang, and Helen Nissenbaum. Unaccounted privacy violation: a comparative analysis of persistent identification of users across social contexts, 2021.

[SD09] Athena Stassopoulou and Marios D. Dikaiakos. Web robot detection: A probabilistic reasoning approach. *Computer Networks*, 53(3), 2009. DOI: 10.1016/j.comnet.2008.09.021.

[SDA+16] Oleksii Starov, Johannes Dahse, Syed Sharique Ahmad, Thorsten Holz, and Nick Nikiforakis. No honor among thieves: A large-scale analysis of malicious web shells. In *Proc. 25th International Conference on World Wide Web (WWW'16)*. ACM, 2016. DOI: 10.1145/2872427.2882992.

[Sel18] Andrew Sellars. Twenty years of web scraping and the computer fraud and abuse act. *BUJ Sci. & Tech. L.*, 24, 2018.

[SHG+18] Quirin Scheitle et al. A long way to the top: significance, structure, and stability of internet top lists. In *Proc. 18th ACM Internet Measurement Conference (IMC'18)*. ACM, 2018. DOI: 10.1145/3278532.3278574.

[SIE+22] Sandra Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. Webgraph: capturing advertising and tracking information flows for robust blocking. In *Proc. 31st USENIX Security Symposium (USENIX Security'22)*. USENIX Association, 2022. ISBN: 978-1-939133-31-1.

[SII+19] Konstantinos Solomos, Panagiotis Ilia, Sotiris Ioannidis, and Nicolas Kourtellis. TALON: an automated framework for cross-device tracking detection. In *Proc. 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID'19)*. USENIX Association, 2019.

[SIK20] Konstantinos Solomos, Panagiotis Ilia, and Nicolas Kourtellis. Clash of the trackers: measuring the evolution of the online tracking ecosystem. In *Proc. 4th Network Traffic Measurement and Analysis Conference (TMA'20)*. IFIP, 2020.

[SK19] Jannick Kirk Sørensen and Sokol Kosta. Before and after GDPR: the changes in third party presence at public and private european websites. In *Proc. 28th The Web Conference (WWW'19)*. ACM, 2019. DOI: 10.1145/3308558.3313524.

[SKD+16] Richard Shay et al. Designing password policies for strength and usability. *ACM Trans. Inf. Syst. Secur.*, 18(4), 2016. DOI: 10.1145/2891411.

[SKP16] Suphannee Sivakorn, Angelos D. Keromytis, and Jason Polakis. That's the way the cookie crumbles: evaluating HTTPS enforcing mechanisms. In *Proc. 15th ACM Workshop on Privacy in the Electronic (WPES'16)*. ACM, 2016. DOI: 10.1145/2994620.2994638.

[Sle17] Marc Sleegers. *Counting sheep: analysing online authentication security*. Bachelor's thesis, Open University of the Netherlands, March 2017. http://www.open.ou.nl/hjo/supervision/2017-marc.sleegers/2017-marc.sleegers-bsc-thesis.pdf.

[SLG19] Michael Schwarz, Florian Lackner, and Daniel Gruss. Javascript template attacks: automatically inferring host information for targeted exploits. In *Proc. 26th Annual Network and Distributed System Security Symposium (NDSS'19)*. The Internet Society, 2019. DOI: 10.14722/ndss.2019.23155.

[SM19a] Takahito Sakamoto and Masahiro Matsunaga. After GDPR, still tracking or not? understanding opt-out states for online behavioral advertising. In *IEEE Symposium on Security and Privacy Workshops*. IEEE, 2019.

[SM19b] Nayanamana Samarasinghe and Mohammad Mannan. Towards a global perspective on web tracking. *Comput. Secur.*, 87, 2019. DOI: 10.1016/j.cose.2019.101569.

[SP13] Abdul Serwadda and Vir V. Phoha. Examining a large keystroke biometrics dataset for statistical-attack openings. *Journla of ACM Transactions on Information and System Security*, 16(2), 2013. DOI: 10.1145/2516960.

[SPK16a] Suphannee Sivakorn, Iasonas Polakis, and Angelos D. Keromytis. I am robot: (deep) learning to break semantic image captchas. In *IEEE European Symposium on Security and Privacy, (EuroS&P'16)*, 2016. DOI: 10.1109/EUROSP.2016.37.

[SPK16b] Suphannee Sivakorn, Iasonas Polakis, and Angelos D. Keromytis. The cracked cookie jar: HTTP cookie hijacking and the exposure of private information. In *Proc. 37th IEEE Symposium on Security and Privacy (SP'16)*. IEEE Computer Society, 2016. DOI: 10.1109/SP.2016.49.

[SPK16c] Suphannee Sivakorn, Jason Polakis, and Angelos D Keromytis. I'm not a human: breaking the google recaptcha. *Black Hat*, 2016.

[SPW+16] Abdul Serwadda, Vir V. Phoha, Zibo Wang, Rajesh Kumar, and Diksha Shukla. Toward robotic robbery on the touch screen. *Journal of ACM Transactions on Information and System Security*, 18(4), 2016. DOI: 10.1145/2898353.

[SRJ+19] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. Don't trust the locals: investigating the prevalence of persistent client-side cross-site scripting in the wild. In *Proc. 26th Annual Network and Distributed System Security Symposium (NDSS'19)*. The Internet Society, 2019. DOI: 10.14722/ndss.2019.23009.

[SRM+17] Grant Storey, Dillon Reisman, Jonathan R. Mayer, and Arvind Narayanan. The future of ad blocking: an analytical framework and new techniques. *CoRR*, abs/1705.08568, 2017. URL: http://arxiv.org/abs/1705.08568.

[Ste18] Thomas Steiner. What is in a web view: an analysis of progressive web app features when the means of web access is not a web browser. In *Proc. 27th The Web Conference (WWW'18)*. ACM, 2018. DOI: 10.1145/3184558.3188742.

[STK17] Peter Snyder, Cynthia Bagier Taylor, and Chris Kanich. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *Proc. 24th ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. ACM, 2017. DOI: 10.1145/3133956.3133966.

[SW19] Youjin Shin and Simon S. Woo. What is in your password? analyzing memorable and secure passwords using a tensor decomposition. In *Proc. 28th The Web Conference (WWW'19)*. ACM, 2019. DOI: 10.1145/3308558.3313690.

[TDK11] Shuo Tang, Nathan Dautenhahn, and Samuel T King. Fortifying web-based applications automatically. In *Proc. 18th ACM conference on Computer and communications security (CCS'11)*. ACM, 2011. DOI: 10.1145/2046707.2046777.

[TJM15] Christof Ferreira Torres, Hugo L. Jonker, and Sjouke Mauw. FP-Block: usable web privacy by controlling browser fingerprinting. In *Proc. 20th European Symposium on Research in Computer Security (ESORICS'15)*, volume 9327 of *LNCS*. Springer, 2015. DOI: 10.1007/978-3-319-24177-7_1.

[TK02] Pang-Ning Tan and Vipin Kumar. Discovery of web robot sessions based on their navigational patterns. *Data Min. Knowl. Discov.*, 6(1), 2002. DOI: 10.1023/A:1013228602957.

[TM21] Giorgio Di Tizio and Fabio Massacci. A calculus of tracking: theory and practice. *Proc. Priv. Enhancing Technol.*, 2021(2), 2021. DOI: 10.2478/popets-2021-0027.

[TS06] Mike Thelwall and David Stuart. Web crawling ethics revisited: cost, privacy, and denial of service. *J. Assoc. Inf. Sci. Technol.*, 57(13), 2006. DOI: 10.1002/asi.20388.

[UDH+20] Tobias Urban, Martin Degeling, Thorsten Holz, and Norbert Pohlmann. Beyond the front page: measuring third party dynamics in the field. In *Proc. 29th The Web Conference (WWW'20)*. ACM, 2020. DOI: 10.1145/3366423.3380203.

[UTD+19] Tobias Urban, Dennis Tatang, Martin Degeling, Thorsten Holz, and Norbert Pohlmann. A study on subject data access in online advertising after the GDPR. In *Data Privacy Management (DPM'19), Cryptocurrencies and Blockchain Technology (CBT'19)*, volume 11737 of *LNCS*. Springer, 2019. DOI: 10.1007/978-3-030-31500-9_5.

[UTD+20] Tobias Urban, Dennis Tatang, Martin Degeling, Thorsten Holz, and Norbert Pohlmann. Measuring the impact of the GDPR on data sharing in ad networks. In *Proc. 15th ACM Asia Conference on Computer and Communications Security (AsiaCCS'20)*. ACM, 2020. DOI: 10.1145/3320269.3372194.

[VAA+21] Yash Vekaria et al. Differential tracking across topical webpages of indian news media. In *Proc. 13th ACM Web Science Conference (WebSci'21)*. ACM, 2021. DOI: 10.1145/3447535.3462497.

[vABH+03] Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. CAPTCHA: using hard AI problems for security. In *Proc. Advances in Cryptology - EUROCRYPT'03, International Conference on the Theory and Applications of Cryptographic Techniques*, 2003. DOI: 10.1007/3-540-39200-9_18.

[VáFG+19] Pelayo Vallina, álvaro Feal, Julien Gamba, Narseo Vallina-Rodriguez, and Antonio Fernández Anta. Tales from the porn: A comprehensive privacy analysis of the web porn ecosystem. In *Proc. 19th ACM Internet Measurement Conference (IMC'19)*. ACM, 2019. DOI: 10.1145/3355369.3355583.

[vAHS17] Steven van Acker, Daniel Hausknecht, and Andrei Sabelfeld. Measuring login webpage security. *Proc. 32nd ACM SIGAPP Symposium On Applied Computing (SAC'17)*, 2017. DOI: 10.1145/3019612.3019798.

[VAW+19] Rob Van Eijk, Hadi Asghari, Philipp Winter, and Arvind Narayanan. The impact of user location on cookie notices (inside and outside of the european union). In *Workshop on Technology and Consumer Protection (ConPro'19). IEEE*. IEEE, 2019.

[vGPJ19] Tom van Goethem, Victor Le Pochat, and Wouter Joosen. Mobile friendly or attacker friendly?: A large-scale security evaluation of mobile-first websites. In *Proc. 14th ACM Asia Conference on Computer and Communications Security (AsiaCCS'19)*. ACM, 2019. DOI: 10.1145/3321705.3329855.

[VHS18] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. Raising the bar: evaluating origin-wide security manifests. In *Proc. 34th Annual Computer Security Applications Conference (ACSAC'18)*. ACM, 2018. DOI: 10.1145/3274694.3274701.

[Vlo18] Gabry Vlot. *Automated data extraction; what you see might not be what you get*. Master's thesis, Open University of the Netherlands, July 2018. http://www.open.ou.nl/hjo/supervision/2018-g.vlot-msc-thesis.pdf.

[VNB+14] Thomas Vissers, Nick Nikiforakis, Nataliia Bielova, and Wouter Joosen. Crying Wolf? On the Price Discrimination of Online Airline Tickets. In *Proc. 7th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs'14)*, 2014.

[VRR+20] Antoine Vastel, Walter Rudametkin, Romain Rouvoy, and Xavier Blanc. FP-Crawlers: Studying the Resilience of Browser Fingerprinting to Block Crawlers. In *Proc. 2nd NDSS Workshop on Measurements, Attacks, and Defenses for the Web (MADWEB'20)*, 2020. DOI: 10.14722/madweb.2020.23010.

[VYG13] Shardul Vikram, Chao Yang, and Guofei Gu. NOMAD: towards non-intrusive moving-target defense against web bots. In *IEEE Conference on Communications and Network Security (CNS'13)*, 2013. DOI: 10.1109/CNS.2013.6682692.

[WD05] Baoning Wu and Brian D. Davison. Cloaking and redirection: A preliminary study. In *Proc. 1st International Workshop on Adversarial Information Retrieval on the Web (AIRWeb'05)*, 2005.

[WD06] Baoning Wu and Brian D. Davison. Detecting semantic cloaking on the web. In *Proc. 15th International Conference on World Wide Web (WWW'06)*, 2006. DOI: 10.1145/1135777.1135901.

[WSL+16] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. CSP is dead, long live CSP! on the insecurity of whitelists and the future of content security policy. In *Proc. 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*. ACM, 2016. DOI: 10.1145/2976749.2978363.

[WSV11] David Y. Wang, Stefan Savage, and Geoffrey M. Voelker. Cloak and dagger: dynamics of web search cloaking. In *Proc. 18th ACM Conference on Computer and Communications (CCS'11)*, 2011. DOI: 10.1145/2046707.2046763.

[WZX+16] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick Eugster. Webranz: web page randomization for better advertisement delivery and web-bot prevention. In *Proc. 24th Foundations of Software Engineering (FSE'16)*. ACM, 2016. DOI: 10.1145/2950290.2950352.

[XLC+18] Haitao Xu et al. Detecting and characterizing web bot traffic in a large e-commerce marketplace. In *Proc. 23rd European Symposium on Research in Computer Security, ESORICS'18*, 2018. DOI: 10.1007/978-3-319-98989-1_8.

[YSM+22] Xiufen Yu, Nayanamana Samarasinghe, Mohammad Mannan, and Amr M. Youssef. Got sick and tracked: privacy analysis of hospital websites. In *EuroS&P Workshops*. IEEE, 2022. DOI: 10.1109/EuroSPW55150.2022.00034.

[YY20] Zhiju Yang and Chuan Yue. A comparative measurement study of web tracking on mobile and desktop environments. *Proc. Priv. Enhancing Technol.*, 2020(2), 2020. DOI: 10.2478/popets-2020-0016.

[ZBO+20] David Zeber et al. The representativeness of automated web crawls as a surrogate for human browsing. In *Proc. 29th The Web Conference (WWW'20)*. ACM, 2020. DOI: 10.1145/3366423.3380104.

[ZE10] Yuchen Zhou and David Evans. Why aren't http-only cookies more widely deployed. *Proc. 4th Web 2.0 Security and Privacy Workshop*, 2, 2010.

[ZE14] Yuchen Zhou and David Evans. SSOScan: automated testing of web applications for single sign-on vulnerabilities. In *Proc. 23rd USENIX Security Symposium (USENIX Security'14)*. USENIX Association, 2014.

[ZJL+15] Xiaofeng Zheng et al. Cookies lack integrity: real-world implications. In *Proc. 24th USENIX Security Symposium (USENIX Security'15)*. USENIX Association, 2015.

[ZOC+21] Penghui Zhang et al. CrawlPhish: Large-scale Analysis of Client-side Cloaking Techniques in Phishing. In *Proc. 42nd IEEE Symposium on Security and Privacy (S&P'21)*, 2021. DOI: 10.1109/MSEC.2021.3129992.

Online References

[ACL22] ACLU. Exhibit 2, signed settlement agreement. 2022. URL: https://www.aclu.org/legal-document/exhibit-2-signed-settlement-agreement. Last access: June 2, 2024.

[AML15] Julia Angwin, Surya Mattu, and Jeff Larson. The tiger mom tax: asians are nearly twice as likely to get a higher price from princeton review. September 2015. URL: https://www.propublica.org/article/asians-nearly-twice-as-likely-to-get-higher-price-from-princeton-review. Last access: June 2, 2024.

[BBC+04] Tim Berners-Lee et al. Architecture of the world wide web, volume one. 2004. URL: https://www.w3.org/TR/webarch/#general. Last access: June 2, 2024.

[BC21] Browser and Platform Configuration. Openwpm. 2021. URL: https://github.com/mozilla/OpenWPM/blob/master/docs/Configuration.md. Last access: June 2, 2024.

[BK23] Mathias Bynens and Peter Kvitek. Chrome's headless mode gets an upgrade: introducing – headless=new. February 2023. URL: https://developer.chrome.com/articles/new-headless/. Last access: June 2, 2024.

[Bla21] Paul E. Black. Ratcliff/obershelp pattern recognition. 2021. URL: https://www.nist.gov/dads/HTML/ratcliffObershelp.html. Last access: June 2, 2024.

[Boo23] Booking.com. Mobile rate. 2023. URL: https://partner.booking.com/en-gb/solutions/mobile-rate. Last access: June 2, 2024.

[But10] Eric Butler. Firesheep. 2010. URL: http://codebutler.com/firesheep/. Last access: June 2, 2024.

[CFA+23] Flavio Copes et al. The v8 javascript engine. 2023. URL: https://nodejs.dev/en/learn/the-v8-javascript-engine/. Last access: June 2, 2024.

[Cim20] Catalin Cimpanu. Apple declined to implement 16 web apis in safari due to privacy concerns. 2020. URL: https://www.zdnet.com/article/apple-declined-to-implement-16-web-apis-in-safari-due-to-privacy-concerns/. Last access: June 2, 2024.

[CNI21] CNIL. Decision of the executive committee of the commission nationale de l'informatique et des libertés n° medp-2021-002 of 6th december 2021 to make public the order n°med-2021-134 of 26th november 2021, issued to clearview ai. 2021. URL: https://www.cnil.fr/sites/default/files/atoms/files/decision_ndeg_medp-2021-002.pdf. Last access: June 2, 2024.

[Com21] Australian Information Commissioner. Commissioner initiated investigation into clearview ai, inc. (privacy) [2021]. 2021. URL: http://www.austlii.edu.au/cgi-bin/viewdoc/au/cases/cth/AICmr/2021/54.html. Last access: June 2, 2024.

[Cou13] Court of Justice. Case c-202/12, innoweb bv v wegener ict media bv and wegener mediaventions bv. 2013. URL: https://curia.europa.eu/juris/document/document.jsf?docid=145914&doclang=EN. Last access: June 2, 2024.

[Cou15] Court of Justice. Ryanair ltd v pr aviation bv. 2015. URL: https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX:62014CJ0030. Last access: June 2, 2024.

[Cou21] Court of Justice. Case c-762/19, cv-online latvia sia v melons sia, 2021. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:62019CJ0762. Last access: June 2, 2024.

[Cro18] Luke Crouch. Data@mozilla: improving privacy without breaking the web. 2018. URL: https://blog.mozilla.org/data/2018/01/26/improving-privacy-without-breaking-the-web/. Last access: June 2, 2024.

[Cyp22] Cypress. Trade-offs. 2022. URL: https://docs.cypress.io/guides/references/trade-offs#Automation-restrictions. Last access: June 2, 2024.

[Duc23] DuckDuckGo. Tracker radar collector. 2023. URL: https://github.com/duckduckgo/tracker-radar-collector. Last access: June 2, 2024.

[Eur19] European Parliament, Council of the European Union. Directive (eu) 2019/790 of the european parliament and of the council of 17 april 2019 on copyright and related rights in the digital single market and amending directives 96/9/ec and 2001/29/ec. 2019. URL: https://eur-lex.europa.eu/eli/dir/2019/790/oj. Last access: June 2, 2024.

[Eur96] European Parliament, Council of the European Union. Directive 96/9/ec of the european parliament and of the council of 11 march 1996 on the legal protection of databases. 1996. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%5C%3A31996L0009. Last access: June 2, 2024.

[Fel20] Pavel Feldmann. How does puppeteer's new firefox support affect playwright? 2020. URL: https://github.com/microsoft/playwright/issues/1765#issuecomment-613178363. Last access: June 2, 2024.

[FGS20] Maja Frydrychowicz, James Graham, and Henrik Skupin. Improving cross-browser testing, part 2: new automation features in firefox nightly. 2020. URL: https://hacks.mozilla.org/2020/12/cross-browser-testing-part-1-web-app-testing-today/. Last access: June 2, 2024.

[FGS21] Maja Frydrychowicz, James Graham, and Henrik Skupin. Improving cross-browser testing, part 2: new automation features in firefox nightly. 2021. URL: https://hacks.mozilla.org/2021/01/improving-cross-browser-testing-part-2-new-automation-features-in-firefox-nightly/. Last access: June 2, 2024.

[GI11] Tali Garsiel and Paul Irish. How browsers work – behind the scenes of modern web browsers. 2011. URL: https://web.dev/howbrowserswork/. Last access: June 2, 2024.

[GPD22] GPDP. Riconoscimento facciale: il garante privacy sanziona clearview per 20 milioni di euro. vietato l'uso dei dati biometrici e il monitoraggio degli italiani. 2022. URL: https://gpdp.it/web/guest/home/docweb/-/docweb-display/docweb/9751362. Last access: June 2, 2024.

[Hel22] Hellenic Data Protection Authority. Imposition of fine on clearview ai, inc - decision 35/2022. 2022. URL: https://www.dpa.gr/en/en/enimerwtiko/prakseisArxis/imposition-fine-clearview-ai-inc. Last access: June 2, 2024.

[Inf22] Information Commissioner's Office. Clearview ai inc. enforcement notice. 2022. URL: https://ico.org.uk/action-weve-taken/enforcement/clearview-ai-inc-en/. Last access: June 2, 2024.

[Int21] Integritetsskyddsmyndighetens. Beslut efter tillsyn enligt brottsdatalagen – polismyndighetens användning av clearview ai. 2021. URL: https://perma.cc/JDC2-48A5. Last access: June 2, 2024.

[Jia21] Li JianTao. You talking to me? 2021. URL: https://starlabs.sg/blog/2021/04-you-talking-to-me/. Last access: June 2, 2024.

[KIZ+22] Martijn Koster, Gary Illyes, Henner Zeller, and Lizzi Sassman. Robots Exclusion Protocol. Internet-Draft draft-koster-rep-12, Internet Engineering Task Force, July 2022. 14 pages. URL: https://datatracker.ietf.org/doc/draft-koster-rep/12/. Work in Progress.

[KM00] D. Kristol and L. Montulli. RFC 2965: HTTP state management mechanism. 2000. URL: https://www.ietf.org/rfc/rfc2965.txt. Last access: June 2, 2024.

[Kos18] Mariko Kosaka. Inside look at modern web browser (part 1). 2018. URL: https://developer.chrome.com/blog/inside-browser-part1/. Last access: June 2, 2024.

[Kos93] Martijn Koster. Guidelines for robot writers. 1993. URL: https://www.robotstxt.org/guidelines.html. Last access: June 2, 2024.

[Kos94] Martjin Koster. A standard for robot exclusion. 1994. URL: http://webdoc.gwdg.de/ebook/aw/1999/webcrawler/mak/projects/robots/norobots.html. Last access: June 2, 2024.

[Lin20] Linder. User-agent switcher. 2020. URL: https://addons.mozilla.org/en-US/firefox/addon/user-agent-switcher-revived/. Last access: June 2, 2024.

[Mer22] Merriam-Webster.com Dictionary. Bot. 2022. URL: https://www.merriam-webster.com/dictionary/bot. Last access: June 2, 2024.

[ner20] neroux. Random user-agent. 2020. URL: https://web.archive.org/web/20210303001755/https://addons.mozilla.org/en-US/firefox/addon/random_user_agent/. Last access: June 2, 2024.

[Off21] Office of the Privacy Commissioner of Canada. Joint investigation of clearview ai, inc. by the office of the privacy commissioner of canada, the commission d'accès à l'information du québec, the information and privacy commissioner for british columbia, and the information privacy commissioner of alberta. 2021. URL: https://www.priv.gc.ca/en/opc-actions-and-decisions/investigations/investigations-into-businesses/2021/pipeda-2021-001/. Last access: June 2, 2024.

[OWA17] OWASP. OWASP top ten – 2017: the ten most critical web application security risks. 2017. URL: https://owasp.org/www-project-top-ten/2017/. Last access: June 2, 2024.

[Rao05] Raosoft. Sample size calculator. 2005. URL: http://www.raosoft.com/samplesize.html. Last access: June 2, 2024.

[Ray20] Ray. User-agent switcher and manager. 2020. URL: https://addons.mozilla.org/en-US/firefox/addon/user-agent-string-switcher/. Last access: June 2, 2024.

[Sch19] Stefan Schwinghammer. Bonkers vs bots: how to get rid of resellers. 2019. URL: https://www.soloskatemag.com/bonkers-vs-bots. Last access: June 2, 2024.

[Sch20] Alexander Schlarb. User-agent switcher. 2020. URL: https://addons.mozilla.org/en-US/firefox/addon/uaswitcher/. Last access: June 2, 2024.

[ser20] sereneblue. Chameleon. 2020. URL: https://addons.mozilla.org/en-US/firefox/addon/chameleon-ext/. Last access: June 2, 2024.

[She15a] Sergey Shekyan. Detecting headless browsers. 2015. URL: https://www.slideshare.net/SergeyShekyan/shekyan-zhang-owasp. Last access: June 2, 2024.

[She15b] Sergey Shekyan. Detecting PhantomJS based visitors. 2015. URL: https://blog.shapesecurity.com/2015/01/22/detecting-phantomjs-based-visitors/. Last access: June 2, 2024.

[Sim23] Similarweb. Browser market share in may 2023. 2023. URL: https://www.similarweb.com/browsers/. Last access: June 2, 2024.

[Sta23] StatCounter. Browser market share worldwide: may 2022 - may 2023. 2023. URL: https://gs.statcounter.com/browser-market-share. Last access: June 2, 2024.

[Sto17] Nikola Stojiljković. Extensions support in headless chrome. 2017. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=706008#c5. Last access: June 2, 2024.

[Sym21] Symantec. Webpulse site review request. 2021. URL: https://sitereview.norton.com/. Last access: June 2, 2024.

[Tea21] OWASP CheatSheets Series Team. Vulnerability disclosure cheat sheet. 2021. URL: https://cheatsheetseries.owasp.org/cheatsheets/Vulnerability_Disclosure_Cheat_Sheet.html. Last access: June 2, 2024.

[Uni06] United States District Court, District of Nevada. Blake a. field v. google inc., no. cv-s-04-0413-rcj-lrl. 2006. URL: https://fairuse.stanford.edu/primary_materials/cases/fieldgoogle.pdf. Last access: June 2, 2024.

[Uni13] United States District Court, District of New York. Associated press v. meltwater u.s. holdings, inc. 2013. URL: https://casetext.com/case/associated-press-v-meltwater-us-holdings-inc. Last access: June 2, 2024.

[Uni19] United States Court of Appeals for the Ninth Circuit. Hiq labs, inc v. linkedin corporation, no. 17-16783, d.c., no. 3:17-cv-03301-emc. 2019. URL: https://cdn.ca9.uscourts.gov/datastore/opinions/2019/09/09/17-16783.pdf. Last access: June 2, 2024.

[Vas17] Antoine Vastel. Detecting chrome headless. 2017. URL: https://antoinevastel.com/bot%20detection/2017/08/05/detect-chrome-headless.html. Last access: June 2, 2024.

[Vas18] Antoine Vastel. Detecting Chrome headless, new techniques. 2018. URL: https://antoinevastel.com/bot%20detection/2018/01/17/detect-chrome-headless-v2.html. Last access: June 2, 2024.

[Vas19] Antoine Vastel. Detecting chrome headless, the game goes on! 2019. URL: https://antoinevastel.com/bot%20detection/2019/07/19/detecting-chrome-headless-v3.html. Last access: June 2, 2024.

[Wei15] Ryan Weinstein. Can a website detect when you are using selenium with chromedriver? 2015. URL: https://stackoverflow.com/questions/33225947/can-a-website-detect-when-you-are-using-selenium-withchromedriver. Last access: June 2, 2024.

[Whi16] White Ops. The methbot operation. 2016. URL: https://www.whiteops.com/methbot. Last access: June 2, 2024.

Artefacts

[ART-CoNEXT22] Benjamin Krumnow, Hugo Jonker, and Stefan Karsch. How gullible are web measurement tools? A case study analysing and strengthening OpenWPM's reliability. 2022. URL: https://bkrumnow.github.io/openwpm-reliability/.

[ART-ESORICS19] Hugo Jonker, Benjamin Krumnow, and Gabry Vlot. Fingerprint surface-based detection of web bot detectors. 2019. URL: https://bkrumnow.github.io/fpbotdetection/.

[ART-IMC21] Daniel Goßen, Hugo Jonker, Stefan Karsch, Benjamin Krumnow, and David Roefs. HLISA. 2021. URL: https://github.com/droefs/HLISA.

[ART-IMC21-2] Daniel Goßen, Hugo Jonker, Stefan Karsch, Benjamin Krumnow, and David Roefs. Spoof navigator.webdriver to false. 2020. URL: https://github.com/openwpm/OpenWPM/pull/526.

[ART-MADWeb23] Hugo Jonker, Benjamin Krumnow, Godfried Meesters, and Stefan Karsch. Diffscraper. 2023. URL: https://github.com/godfriedmeesters/diffscraper.

Appendix

Appendix A

Default Search Settings in Price Differentiation Study

For our study described in Section 5.3, we compared the default settings in each store's search mask. The results are summarised in Tbl. A.1. We found additional services that can be selected within the search mask when using AirFrance. This option is disabled by default on desktops, and it is unavailable in the app. Eurowings is the only vendor that does not offer fare selection via its search mask. Instead, we ensured to use prices shown on the result pages, which always used the lowest fare in our cases. For Opodo, we found that users can select between different restrictions on stops on their trips. However, no restriction is the default option for all checked store.

Table A.1: Overview of default settings in evaluated search masks

Airline	Store	Route	Travellers	Fare	Custom options
AirFrance	.de	round-trip	1 adult	economy	bluebiz off
	.fr	round-trip	1 adult	economy	bluebiz off
	app	round-trip	1 adult	economy	n/a
Eurowings	.de	one-way	1 adult	n/a	n/a
	.fr	one-way	1 adult	n/a	n/a
	app	one-way	1 adult	n/a	n/a
Expedia	.de	round-trip	1 adult	economy	n/a
	.fr	round-trip	1 adult	economy	n/a
	app	round-trip	1 adult	economy	n/a
Kayak	.de	round-trip	1 adult	economy	0 bags
	mob.	round-trip	1 adult	economy	0 bags
	app	round-trip	1 adult	economy	0 bags
Opodo	.de	round-trip	1 adult	economy	No stop restrictions
	.fr	round-trip	1 adult	economy	No stop restrictions
	app	round-trip	1 adult	economy	No stop restrictions

Appendix B

Notes on HLISA

Section B.1 provides an overview of all events we identified as possible candidates to measure interaction (see Section 8.3). In Section B.2, we give a full API specification for HLISA, corresponding to our description from Section 8.4.

B.1 Events related to or triggered by interaction

- **Document:**
  - `copy`
  - `cut`
  - `dragend`
  - `dragenter`
  - `dragleave`
  - `dragover`
  - `dragstart`
  - `drag`
  - `drop`
  - `fullscreenchange`
  - `gotpointercapture`
  - `keydown`
  - `keypress`
  - `keyup`
  - `lostpointercapture`
  - `paste`
  - `pointercancel`
  - `pointerdown`
  - `pointerenter`
  - `pointerleave`
  - `pointermove`
  - `pointerout`
  - `pointerover`
  - `pointerup`
  - `scroll`
  - `selectionchange`
  - `selectstart`
  - `touchcancel`
  - `touchend`
  - `touchmove`
  - `touchstart`
  - `transitionend`
  - `transitionrun`
  - `transitionstart`
  - `visibilitychange`
  - `wheel`
- **Window:**
  - `resize`
  - `focus`
- **Element:**
  - `auxclick`
  - `blur`
  - `click`
  - `contextmenu`
  - `dblclick`
  - `focusin`
  - `focusout`
  - `focus`
  - `mousedown`
  - `mouseenter`
  - `mouseleave`
  - `mousemove`
  - `mouseout`
  - `mouseover`
  - `mouseup`
  - `select`

B.2 HLISA API description

Table B.1: The HLISA API

API function	**Args**	**Description**
HLISA_ActionChains()	webdriver	Constructor to create an action chain
perform()		Executes actions in a chain
reset_actions()		Removes all actions from the current chain
pause()	duration	Pauses the execution of the action chain (in sec)
move_to()	x,y	Moves the cursor from the current position to a given position
move_by_offset()	x, y	Moves the cursor relative to the current position
move_to_element()	element	Moves the cursor to a position within an element's boundaries
move_to_element_with_offset()	element, x, y	Moves the cursor relative to an element's top-left corner
move_to_element_outside_viewport()	element	Scrolls element into the viewport before using `move_to_element`
click()	element	Clicks. If element is provided, first performs `move_to_element`
click_and_hold()	element	Same as `click` without release action
release()	element	Same as `click` without press action
double_click()	element	Same as `click` with an additional click shortly after the first
send_keys()	keys	Executes a human typing rhythm for the given keys
send_keys_to_element()	element, keys	Selects the element, then executes the `send_keys` function
scroll_by()	x, y	Scrolls the viewport till a distance is taken
scroll_to()	x, y	Scrolls until the specified position is in the top left corner
context_click()	element	Same as `click` using a right mouse button
drag_and_drop()	element1, element2	Press left button over element1, move mouse to element2, release mouse button
drag_and_drop_by_offset()	element, x, y	Press left mouse button on element, moves to target offset (x, y) and releases button

functionname() replacement for Selenium function to interact human-like
functionname() new function, not available in Selenium
functionname() passthrough to Selenium's implementation

Appendix C

OpenWPM in Literature

The following provides additional material to our literature study from Section 9.2. We list venues where OpenWPM-based studies were published, including our own work, in Tbl. C.1. Tbl. C.2 provides a detailed view on studies that we considered during our analysis. Each category that applies to a study is marked with a "✓". For those studies that measure certain aspects, but rely on out of bound mechanisms (e.g., by deploying a proxy) and do not rely on OpenWPM's instrumentation are marked with a "○". Running modes are shortened in the table as follow: unspecified (u), native (n), headless (h), xvfb (x), docker (d), virtual machine (v). Papers that are not included in the seed list, but where added by us, are highlighted with a "⋆". Studies marked with a "†" use an OpenWPM data set, but do not perform their own data acquisition.

Table C.1: Publications using OpenWPM aggregated by year and venue

2014	
CCS	1
CoSN	1

2015	
WWW	1
NDSS	1
Tech Science	1
W2SP	1

2016	
IFIP AICT	1
CCS	1
WWW	1

2017	
NDSS	1
PETS	1
CODASPY	1
IWPE	1
Annual Privacy Forum	1
USENIX	1
Appl. Econ. Letters	1

2018	
PETS	2
ACM TOIT	1
CCS	1
ACSAC	1
AINTEC	1

2019	
DPM	2
IDCLR	1
WorldCIST	1
ConPro	1
WWW	1
EuroS&P	1

2020	
PETS	5
WWW	4
TMA	2
ASIACCS	1
PAM	1
EuroS&PW	1
PrivacyCon	1
EuroS&P	1
GLOBECOM	1

2021	
PETS	4
NDSS	1
S&P	1
IMC	1
WebSci	1
IEEE TSNM	1

2022	
PETS	3
USENIX	3
WWW	2
EuroS&PW	1
CoNEXT	1

Table C.2: Overview of previous studies using OpenWPM for web studies

		uses		measures/analyses			performs			visits	uses	mentions
Year	**Ref.**	**Mode**	**VM**	**Cookies**	**HTTP**	**JS**	**Scrolling**	**Clicking**	**Typing**	**Sub-pages**	**Anti-BD**	**BD**
2014	[AEE+14]	u	✓	○	○	✓						
	[RB14]	u						✓	✓			
2015	[ERE+15]	u		✓	✓							
	[KB15]	u		✓	○							
	[AGH15]	h		✓	✓			✓		✓		
	[FMS+15]	u	✓	✓	✓							
2016	[AJ16]	u		✓	✓					✓		
	[EN16]	x	✓	✓	✓	✓				✓		
	[SDA+16]	u	✓		✓							
2017	[MSN17]	u	✓		○	✓		✓				
	[BRA+17]	u	✓	✓	✓							
	[RK17]	u			✓							
	[OEN17]	u				✓						
	[MWP+17]	u		✓	✓							
	[LWP+17]	h										
	[Sch17]	u			✓							
2018	[GKR+18]	u			✓							
	[EHN18]	u			✓			✓	✓	✓		✓
	[BZK+18]	h		✓	✓							
	[DAB+18]	u			✓	✓						✓
	[VHS18]	u			✓							
	[DMF18]	u			✓							
2019	[CHP+19]	u					✓	✓	✓	✓		
	[VAW+19]	d										
	[SK19]	u	✓		✓					✓		
	[LLZ+19]	u			✓							✓
	[MAF+19]	u			✓			✓		✓		
	[RUW19]	0										
	[MGF19]	u			✓							
	[AOM+19]	u		✓								
	[SM19b]	u		✓	✓							✓
	[MSH19]	u			✓							
	[SII+19]	u					✓	✓				
	[VáFG+19]	u		✓	✓	✓						
	[ESORICS19]	h		✓		○						✓
	[UTD+19]	u		✓	✓					✓		
	[SM19a]	u		✓								
2020	[FBL+20]	u		✓	✓					✓		
	[CNS20]	u					✓				✓	✓
	[YY20]	u		✓	✓	✓	✓					
	[AEN20]	u	✓		✓	✓				✓	✓	✓
	[KTK20]	d		✓	✓	✓		✓			✓	
	[ZBO+20]	x	✓	✓	✓	✓						✓
	[ADZ+20]	u			✓						✓	✓
	[AJP+20]	h	✓	✓	✓	✓						
	[UDH+20]	u		✓	✓	✓	✓			✓	✓	✓
	[UTD+20]	u	✓	✓	✓		✓			✓	✓	✓
	[PDA+20]	u		✓								
	[FSK+20]	u		✓	✓							
	[SCL+21]	u	✓	✓	✓	✓					✓	
	[HdTS20]	u		✓	✓							
	[DF20b]	u			✓							
	[SIK20]	n		✓	✓							
	[DF20a]	u			✓							
2021	[CUT+21]	u		✓	✓					✓		✓
	[RTM21]	u	✓		✓	✓				✓		
	[IES21]	u			✓	✓						
	[IMC21]	n			✓		✓	✓	✓			✓
	[TM21]	u			✓							
	[RM21]	u			✓							
	[HDU+21]	u										
	[VAA+21]	u		✓	✓	✓				✓		
	[DMF21]	u			✓			✓				
2022	[CLB+22]	u			✓	✓						✓
	[SIE+22]	u		✓	✓	✓						
	[IWN+22]	u		✓	✓	✓		✓		✓		✓
	[FSL+22]	u		✓	✓	✓						
	[DGU+22]	n/h	✓		✓				✓	✓		✓
	[YSM+22]	h		✓	✓	✓						
	[MN22]	u			✓						✓	✓
	[SAM+22]	u	✓	✓	✓	✓						
	[BKC+22]	u	✓	✓			✓			✓		

List of Figures

List of Tables

Index

The index below highlights explanations of concepts.