Technische Universität Braunschweig

Institute for Application Security

Advanced Attack and Vulnerability Scanning for the Modern Web

Von der
Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines
Doktoringenieurs (Dr.-Ing.)

genehmigte Dissertation

von
Marius Musch

Eingereicht am:	7. Juli 2022
Disputation am:	25. November 2022
1. Referent:	Prof. Dr. Martin Johns
2. Referent:	Prof. Dr. Nick Nikiforakis

2022

Abstract

Today, the Web is at the center of our digital society. Unfortunately, this omnipresence also makes it a worthwhile target for attacks. Thus, security testing should be part of every web development project to identify vulnerabilities before attackers do so. To support this process, this thesis covers advanced attack and vulnerability detection techniques as part of a security scanner for the modern Web. For this, we focus on automated black-box dynamic analyses, as manual work would not scale to the size of the Web platform. We introduce four major web development trends that make scanning modern websites more challenging: complex client-side code, a blurring of involved parties, volatile content, and the use of emerging features. Consequently, we show how a modern security scanner can overcome these challenges by integrating an instrumented browser.

Throughout this thesis, we present three real-life use-cases for our scanner and conduct empirical analyses on a scale of millions of websites to demonstrate its ability to detect attacks and vulnerabilities in the wild. First, we show how our scanner can be used in a preventive manner, i.e., by determining the compatibility of a defensive mechanism on websites that are not yet vulnerable. Second, we present an automated methodology to detect anti-debugging techniques that try to hinder the manual analysis of a website. Third, we investigate the abuse of WebAssembly as part of the cryptojacking phenomenon and also as a new way to write evasive malware for the Web. However, the very same technology that enables these accurate security scans can also introduce new vulnerabilities for the unwary. To account for this, we conclude this thesis with an additional study on the potential danger of using an instrumented browser within publicly exposed web applications.

Zusammenfassung

Das Web spielt heutzutage eine zentrale Rolle in unserer digitalen Gesellschaft. Leider macht diese Omnipräsenz es auch zu einem lukrativen Ziel für Angriffe. Um Verwundbarkeiten zu identifizieren bevor ein Angreifer diese findet, sollte das Prüfen auf Sicherheitslücken ein fester Bestandteil während der Entwicklung von Webanwendungen sein. Um diesen Prozess zu unterstützen, behandelt diese Dissertation die fortgeschrittene Erkennung von Angriffen und Verwundbarkeiten im Rahmen eines Sicherheitsscanners für das moderne Web. Dabei fokussieren wir uns auf automatische, dynamische Black-Box-Analysen, da manuelle Arbeit für die Größe der Web-Plattform nicht angemessen wäre. Wir stellen vier wichtige Webentwicklungs-Trends vor, welche das Scannen auf modern Webseiten anspruchsvoll machen: Komplexer Code auf der Client-Seite, ein Verschwimmen der involvierten Parteien, wechselhafte Inhalte und die Verwendung von neu entstehenden Funktionalitäten. Darauffolgend zeigen wir, wie ein moderner Sicherheitsscanner diese Herausforderungen überwinden kann, in dem ein instrumentierter Browser integriert wird.

Im Laufe dieser Dissertation stellen wir dann drei realistische Anwendungen für unseren Scanner vor und führen empirische Studien in der Größenordnung von Millionen an Webseiten durch, was die Fähigkeit des Scanners Angriffe und Verwundbarkeiten in freier Wildbahn zu finden demonstriert. Zuerst zeigen wir wie unser Scanner als Präventivmaßnahme verwendet werden kann, indem wir die Kompatibilität eines Verteidigungsmechanismus auf Webseiten, die noch nicht verwundbar sind, messen. Danach präsentieren wir eine vollautomatische Methodik um Anti-Debugging-Techniken zu finden, die versuchen die manuelle Analyse von Webseiten zu unterbinden. Als Drittes untersuchen wir den Missbrauch der WebAssembly-Technologie als Teil des sogenannten Cryptojackings, sowie als neuen Weg um evasive Schadsoftware für das Web zu programmieren. Allerdings kann genau dieselbe Technologie, die es uns ermöglicht diese akkuraten Sicherheitsscans zu realisieren, auch in neue Verwundbarkeiten für Unachtsame resultieren. Um dies zu berücksichtigen, endet diese Dissertation mit einer zusätzlichen Studie über die potentiellen Gefahren bei der Verwendung von instrumentierten Browsern im Rahmen von öffentlich zugänglichen Web-Anwendungen.

Acknowledgments

First of all, I want to thank my advisor Martin Johns for his unwavering support over these five years of my PhD. Starting as the very first member of our basically non-existing team, I had no idea what to expect. And, oh boy, what a journey this has been. Throughout the whole time, if something important came up I knew I could always count on your support, and this was true for both academic and personal matters. Moreover, I want to thank Nick Nikiforakis for taking his time to be my second advisor and Wolf-Tilo Balke for chairing the defense.

Next, I want to say a huge thanks to the whole IAS, you are what made this journey bearable through the many discussions, rants, breaks and overall fun times: Alexandra Dirksen, Jannik Hartung, Tobias Jost, Manuel Karl, David Klein, Robin Kirchner, Simon Koch, and Robert Michael. I really like how many of us are not just colleagues, but have grown to become great friends over the years. Special shout-out to my favorite chat group of all time, the legendary IASpam.

I want to thank Konrad Rieck for hosting me while IAS was still in its inception, and for being a mentor the whole time after. Likewise, I am grateful to the former SEC team for a warm welcome during my first six months in Braunschweig: Daniel Arp, Hugo Gascon, Ansgar Kellner, Alwin Maier, Erwin Quiring, and Frank Rust. In particular, I want to thank Christian Wressnegger for his guidance and the resulting collaborations.

I am grateful to Ben Stock, Marius Steffens, and Sebastian Roth from CISPA for getting me started on my first paper and want to thank Sebastian Lekies, Nicolas Golubovic, and the whole ISA team at Google for a great summer in Zurich during my internship in 2019. What connects these two groups is their ties back to Mannheim, so a huge thanks to the CTF team squareroots for getting me interested in IT security in the first place.

Moreover, a shout-out to everyone from both academia and industry who helped or inspired me on this journey: Frederik Braun, Christian Dresen, Tom Van Goethem, Samuel Groß, Thorsten Holz, Christoph Kerschbaumer, Michael Kümmeling, Guoli Ma, Tobias Müller, Juan David Parra Rodriguez, Michael Rodler, Sebastian Schinzel, Mike West, John Wilander, Henrik Willert, and sorry if I forgot about you, dear reader.

I am also grateful to Maren Jansen, Reni Klein, Florian Loch, Fabian Mack, Lea Steffen, and Michael Wagner for being such awesome friends. Last but not least, I want thank my family: My parents and grandparents for putting me on the right path, and my siblings who I can proudly watch finding their own path. Thanks for always being there for me.

Publications

This thesis contains ideas and results of research projects that have been led by the author of this thesis. All of them have been accepted at renowned peer-reviewed security conferences:

- [177] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. "ScriptProtect: Mitigating Unsafe Third-Party Javascript Practices". In: *Proc. of ACM Asia Conference on Computer and Communications Security* (ASIA CCS). 2019.
- [178] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. "New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild". In: *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment* (DIMVA). 2019.
Best Paper Award Runner-up
- [179] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. "Thieves in the Browser: Web-based Cryptojacking in the Wild". In: *Proc. of International Conference on Availability, Reliability and Security* (ARES). 2019.
Best Paper Award Runner-up
- [175] Marius Musch and Martin Johns. "U Can't Debug This: Detecting JavaScript Anti-Debugging Techniques in the Wild". In: *Proc. of USENIX Security Symposium*. 2021.
- [176] Marius Musch, Robin Kirchner, Max Boll, and Martin Johns. "Server-Side Browsers: Exploring the Web's Hidden Attack Surface". In: *Proc. of ACM Asia Conference on Computer and Communications Security* (ASIA CCS). 2022.

Furthermore, selected contributions by the author during the following research collaborations are included in this thesis as well:

- [238] Marius Steffens, Marius Musch, Martin Johns, and Ben Stock. "Who's Hosting the Block Party? Studying Third-Party Blockage of CSP and SRI". In: *Proc. of Network and Distributed System Security Symposium* (NDSS). 2021.
- [128] Manuel Karl*, Marius Musch*, Guoli Ma, Martin Johns, and Sebastian Lekies. "No Keys to the Kingdom Required: A Comprehensive Investigation of Missing Authentication Vulnerabilities in the Wild". In: *Proc. of Internet Measurement Conference* (IMC). 2022.

*Both authors contributed equally

Contents

1 Introduction

I use maps to find out where explorers have already been. Then I go the other way.

— Javad Nasrin, Ondu relic hunter

1.1 Motivation

What started as a new way for scientists to share information at CERN in 1989 turned out to be one of the most important inventions of the 20th century [253]. Today, *the Web* allows us to access information from hundreds of millions of websites from all over the world [183], with the very largest websites each receiving over one billion visits per day [184]. Moreover, the Web not only connects us to servers that provide information but also people regardless of how far away they are in the physical realm. This means we can use it to stay in touch with friends and loved ones, to work remotely without missing a meeting, to transfer money and digital goods, to access the largest collection of knowledge in human history, to share our favorite cat videos [277], and so much more.

However, the omnipresence of the Web and its ever-increasing relevance for commerce and finance also makes it a lucrative environment for malicious actors. For example, in 2018 the booking website of British Airways was modified by hackers to exfiltrate all customer information as it was entered into the page, resulting in a breach that affected over 400,000 customers and a fine of 20 million GBP for the company [18]. Unfortunately, this is not just an isolated incident, but rather only the tip of the iceberg. At the time of writing, the *Have I Been Pwned* project lists data breaches of 580 popular websites that resulted in over 11 billion compromised accounts [112]. While such leaks often get a lot of media attention, there are also targeted attacks on the other end of the spectrum, of which only little is known. Google's Project Zero found that about 50% of all 0-day exploits that were discovered in the wild in 2020 and 2021 do target a web browser [205], again highlighting the importance of the Web as an attack vector.

Writing completely secure code in the first place would be ideal but is a utopian goal as humans are bound to make mistakes, especially given the high complexity of software projects, e.g., browsers with over 25 million lines of code [23]. A widely popular approach to nevertheless prevent vulnerabilities is to employ *security testing* of a software product, such as a web application or browser. In contrast to *software testing*, which evaluates the functionality of a software product for a given purpose, security testing is concerned with evaluating non-functional properties such as confidentiality, integrity, and availability [75]. Security testing consists of techniques that can be classified based on multiple criteria: *White-box testing* has access to design documents and the source code of the product and thus tests from an internal point of view, while *black-box testing* has no such information and only relies on the observable behavior of the software [25]. *Static testing* relies on access to design or code artifacts to check the product without executing the code, while *dynamic testing* evaluates the software by observing its in- and outputs during execution [8]. Moreover, the level of automation is another important aspect, with *manual testing* by a human on one side and fully *automated testing* on the other [12].

Since artifacts such as code, models, and annotations are often unavailable in practice [71], we focus on black-box and consequently dynamic security testing in this thesis. This approach emulates an external hacker and is thus standard practice for security penetration tests [200]. In contrast to hired penetration testers who spend multiple days on a single web application, we as researchers are instead interested in analyzing millions of websites to reason about the whole ecosystem and make generalized claims. Therefore, we focus on completely automated testing for attacks and vulnerabilities in the form of a *security scanner* that was developed as part of this thesis. However, automatically detecting vulnerabilities in modern web applications has become increasingly hard and black-box scanning is especially challenging [71]. In the next chapter, we give an overview of these challenges and describe web development trends that further complicate analyses. First, however, we summarize our thesis contributions and outline the overall structure.

1.2 Thesis Contributions

In this thesis, we discuss automated attack and vulnerability detection techniques in the context of a black-box security scanner for the modern Web. This focus on automation and not requiring access to the server-side source code allows us to conduct empirical studies on a scale of millions of websites. In particular, we make the following contributions:

Advancing the state of scanning We take a look at traditional web scanning and, in a collaboration with Google, create a fast and scalable pipeline based on their Tsunami scanner. We subsequently use this pipeline to scan all websites in the whole IPv4 space in a single day. From there, we discuss how to generalize such scans to support whole vulnerability classes and the resulting challenges. In particular, we focus on four web development trends that further complicate the security scanning of modern websites and necessitate novel solutions. Using the example of SMURF, we then showcase how to overcome one of these challenges by integrating a real browser controlled by our customized instrumentation code (Chapter 2).

Preventing Client-Side XSS We demonstrate how to use a security scanner in a *preventive* manner, i.e., instead of searching for existing vulnerabilities, we measure the compatibility of a defensive mechanism on websites that are not yet vulnerable. For this, we first develop a novel defense called ScriptProtect that protects websites from benign-but-buggy third-party integrations. Our approach transparently wraps dangerous JavaScript functions and prevents misuse by third parties, without affecting many benign use cases. We measure usage of these APIs on real websites and use this to predict the compatibility of our approach with existing web applications, as seamless support of legacy applications was one of our major design goals. Thereby, we conclude that a compromise between security and compatibility is sometimes necessary and create a version of ScriptProtect that works out-of-the-box on 30% of all websites while still preventing 90% of the attacks (Chapter 3).

Discovering anti-debugging techniques We present our work under a new threat model that considers attackers who want to prevent the *manual analysis* of a website through the integrated *Developer Tools* of the browser. To show that this is already happening in the wild, we design an automated approach to detect these attacks with our security scanner. First, we introduce 6 basic *anti-debugging* techniques and then 3 additional, sophisticated techniques, which are based on side-channels. To detect the latter, we rely on a replay system to compare multiple executions of the same website in different environments, thereby facing the challenge of *volatile content* in websites. We then use our detection methodology in the first large-scale study on this issue and report our results. In particular, we investigate websites that simultaneously use multiple of these techniques in more detail (Chapter 4).

Studying malicious WebAssembly We showcase how a modern security scanner is ideally suited to detect completely new attacks that make use of *emerging technologies*. For this, we design a detection methodology for *cryptojacking* attacks, which abuse the computational efficiency provided by the new WebAssembly language to mine cryptocurrency directly in the browser. Our approach uses three phases that at its core rely on a sampling-based profiler to detect hot functions that correspond to mining activity. As one result of our study of cryptojacking attacks in the wild, we show that our three-phase approach significantly outperforms the static approaches used by existing defenses. Moreover, based on a manual analysis of the WebAssembly ecosystem, we present the first evidence that this new language is also already used *evasively*, i.e., by hiding JavaScript malware inside WebAssembly modules (Chapter 5).

Revealing insecure automated browsers All our previous contributions relied on using an *instrumented browser* as part of an automated security scanner. In our final study, we investigate the use of such automated browsers on the *server-side*, where they represent a unique threat by exposing a wide attack surface. We present a scanning methodology that uncovers these automated browsers and use a combination of fingerprinting and timing information to learn more about them. While the total number of automated browsers is moderate, the ones we find are often running severely outdated versions. Remarkably, the majority of them had not been updated for more than 6 months and over 60% of the discovered implementations were found to be vulnerable to publicly available proof-of-concept exploits (Chapter 6).

1.3 Thesis Overview

In the remainder of this thesis, we discuss approaches to detect vulnerabilities and attacks as part of a modern security scanner. In the next chapter, we first introduce the required background knowledge and foundation required for the rest of this thesis. The following three chapters then present three use cases for such a modern security scanner. Then, we conduct one final study on the potential drawbacks of utilizing instrumented browsers on the server-side. We conclude with a summary of this thesis and an outlook.

Chapter 2 motivates the need for modern web security scanning approaches to detect vulnerabilities as well as ongoing attacks. First, we take a look at an exemplary scanning pipeline and discuss the scanning and analysis challenges we will run into when trying to expand its capabilities. Moreover, we introduce four web development trends that further complicate such scanning endeavors on the Web, thus necessitating further research in this area. Finally, we show how to incorporate an automated browser into a security scanner as one part of the solution and discuss the various instrumentation approaches.

Chapter 3 demonstrates how such a modern web security scanner can be used in a preventive manner. For this, we design a backward-compatible defensive mechanism that protects against benign-but-buggy third-party JavaScript code. We then use our modern scanner to evaluate the effectiveness and compatibility of this mechanism, in particular by accurately tracking inclusion relations and code provenance.

Chapter 4 shows how such a modern scanner can also detect subtle attacks against the manual analysis of websites. We create a methodology to automatically detect anti-debugging techniques implemented in JavaScript, with a focus on advanced techniques that rely on side-channels and are thus especially difficult to detect. We present a systematization of techniques and then conduct a large-scale study on their prevalence and severity.

Chapter 5 discusses the difficulty of dealing with a platform that is constantly evolving, as the introduction of new features can also result in new capabilities for attackers. We present two studies on the ecosystem of malicious WebAssembly, which relatively new addition to the browser and the first time that another programming language besides JavaScript was introduced.

Chapter 6 diverges from the previous chapters by not introducing yet another use case for modern security scanning. Instead, it shows the potential drawbacks of automated systems that incorporate a full browser. For this, we conduct a study on outdated automated browsers in the wild that visit arbitrary URLs and are vulnerable to known exploits.

Chapter 7 summarizes our results and contributions. Moreover, it also provides an outlook on potential future research endeavors.

2 Web Security Scanning

In this chapter, we will now take a closer look at how dynamic analysis techniques can be implemented for web technology. For this, we first briefly describe how a typical web security scanning pipeline works in Section 2.1. Then, we look into one specific use case in detail and discuss the challenges we would run into if we would want to extend this specific scanner into a generic one. In Section 2.2, we then expand on this idea of a generic web vulnerability scanner and investigate four web development trends and the challenges they cause for scanning on the modern Web. These four trends are: a heavy reliance on JavaScript code, a blurring of the involved parties due to script inclusions, a general non-deterministic behavior on page load, and the use of emerging browser features accessible via JavaScript APIs. Finally, in Section 2.3 we discuss how to overcome these challenges by integrating a real browser into our security scanner. For this, we describe several browser instrumentation approaches, including browser extensions, the Chrome DevTools Protocol (CDP), and native instrumentation. We discuss the advantages and drawbacks of the respective approaches and conclude with a showcase of how to solve one of the previously outlined challenges, in this case the blurred parties, with the CDP instrumentation approach.

2.1 Traditional Web Scanning

Imagine a new critical vulnerability for a web application is publicly disclosed and you are tasked with designing a tool that discovers all affected machines in a given large network. On a high level, you likely come up with a process that looks roughly as follows: As a first step, you either need an already existing and up-to-date list of all assets in the network, or you need to discover all hosts that are online yourself, e.g., through a port scan. Next, you have to determine which of these hosts are actually running a web server and thus speak HTTP(S). Then, you might want to determine if the discovered web server is in scope, which means making certain it actually runs the affected web application before probing further. But at this point, it is still not clear if the server is running an outdated and insecure version, or if the latest security patches were already applied. Therefore, we need the crucial step that determines whether a vulnerability is present but without causing any harm. Depending on the complexity of the application and the particular vulnerability this might be achieved with one single request or might require multiple, subsequent requests.

So far, this description was only an abstract overview of the different stages of a security scanning pipeline. In the following subsection, we will go into more detail by examining the scanning pipeline that we used to check the whole routed IPv4 space, i.e., roughly

3.5 billion IP addresses, for the presence of 18 different pre-authentication vulnerabilities *within a single day*. After that, we will discuss some new scanning challenges that emerge when the goal is to detect *generic vulnerabilities*, i.e., when the details such as vulnerability location and context are not known beforehand. Then, we describe potential solutions in the form of *emulated browsers*, which support many more features than our presented scanning pipeline based on plain HTTP requests.

2.1.1 Exemplary Pipeline: Tsunami

Nowadays, applications expose administrative endpoints to the Web that can be used for a plethora of security-sensitive actions. Typical use cases range from running small snippets of user-provided code for rapid prototyping to managing job scheduling on whole clusters of computing devices. While such web applications make the lives of administrators easier, they can be leveraged by attackers to compromise the underlying infrastructure. This is especially true because many of these applications have been designed for usage within an intranet or on localhost and thus are lacking strong authentication mechanisms. However, with the advent of cloud computing, more and more of these applications are getting deployed to the cloud and need to be accessed through the Web. Due to the lack of authentication mechanisms, it is dangerous to expose such applications to the Internet. For example, an exposed admin panel of a continuous integration server can be abused by an attacker to push a malicious binary to a build server.

In our paper "No Keys to the Kingdom Required" [128] we investigate the prevalence of what we called *missing authentication vulnerabilities (MAVs)*. For this, we first selected 25 popular applications that contain an *administrative web endpoint (AWEs)*, manually analyzed them, and found that 18 of the 25 endpoints are in scope for our study on MAVs. We set up to conduct a large-scale study that aims to determine the prevalence and longevity of these 18 MAVs in the wild. To perform this study on the whole IPv4 address space in a reasonable time frame, we built a custom scanning pipeline consisting of four stages based on existing open-source tools as well as specialized components of our own. In what follows, we describe each stage and its components in more detail, serving as one concrete example of how the abstract scanning pipeline from the beginning of this section can be applied in one specific scenario.

Stage I – Discovery As the first step, we checked which hosts are actually online and expose a service to the Internet by running a port scan. For this, we made use of the existing tool *Masscan* [93], which is an extremely fast port scanner written in C. We excluded all IANA reserved allocations [113] from our scan, such as those reserved for Multicast, private use, or the US Department of Defense, leaving us with roughly 3.5B IPv4 addresses to scan. As an Internet-wide scan produces a large amount of traffic, we limited our scan to the 12 most important ports for our study: 80, 443, and all default ports of the 18 selected applications, which have some overlap. To prevent using stale results, i.e., running the next two stages on hosts that went offline in the meantime, we alternated between these three stages by only scanning a tiny fraction of all hosts and immediately following up with the other stages before we continued the port scan. In this stage, we already could reduce the number of potential scanning targets from 42B (3.5B hosts * 12 ports) to about 165M ports that are actually open and need further scanning with our subsequent stages.

Stage II – Prefilter For all open ports identified by the first stage, we then checked if they speak HTTP and/or HTTPS, otherwise we discarded them as out of scope. If one or both connections succeeded, we followed redirects until we received a response body and searched it with our *prefilter signatures*. These signatures are short regular expressions, which indicate that the response was generated by one of the 18 applications of our study. We crafted these signatures manually, by looking for unique keywords or snippets that stayed stable across many different versions of the applications, e.g., `wp-json` for WordPress, `name="Generator" content="Drupal` for Drupal, and `/static/yarn.css` for Hadoop. In total, we created 90 such signatures, an average of 5 per application. These signatures do not determine if the detected applications are vulnerable, but instead reduce the number of targets to scan in the third stage from about 103M ports speaking HTTP(S) to about 2.5M targets running one of the 18 AWEs in scope for our study. We implemented this prefilter in Node.js using the popular `requests` package to handle the underlying HTTP communication including the redirects. Our signatures are applied as a simple string search on the response body, so we do not need to parse and render the response and also do not need to request embedded resources like scripts and stylesheets. Thus, we can run this stage easily for hundreds of targets in parallel on a single machine without performance issues.

Stage III – Tsunami At this point, we confirmed that the application is in scope for our study and now need to verify whether it is running in an insecure state due to a MAV. For this, we created and open-sourced a generic network security scanner called *Tsunami* [86]. It has an extensible plugin system and each MAV verification logic is implemented as a dedicated Tsunami plugin. Based on the port and application information we had already collected from *Stage I* and *Stage II*, Tsunami selects the appropriate MAV detection plugins for each matching application. For example, in order to check whether a WordPress installation can be hijacked, we have a plugin that queries the `/wp-admin/install.php` page and checks whether the WordPress installation process was served on that link. Compared to the simple signatures of the prefilter, these plugins can involve more complex logic, e.g., a verification that involves two or more requests or one that involves extracting information first and then conditionally acting on that. Under the hood, the Tsunami scanner is written in Java and uses a custom HTTP client based on the `OkHttp` library. Like the prefilter, it supports basic HTTP features like redirects and POST requests, but for performance reasons does not try to parse or even render HTML responses.

Stage IV – Fingerprinting To enable detailed analyses of the discovered AWEs, we added an additional step that goes beyond vulnerability verification and also tries to determine the exact version of the deployed applications. This way, we can infer their default settings, which could have changed over time, as well as their up-to-dateness in general. Manual inspection showed that 13 out of the 18 applications reveal their version number somewhere on the website even to unauthenticated requests. For example, Kubernetes reveals the version when requesting the `/version` API endpoint, while Consul includes an HTML comment with the version on the front page. Therefore, our version fingerprinting first tries to extract the exact version number from the 13 applications where this information is usually voluntarily revealed on the page or in the HTML source.

For the five remaining applications, as well as cases where this version number was removed, we employ a more elaborate fingerprinting mechanism. This fingerprinter is based on two major components, a *knowledge base* and a *crawler*, and implemented as an additional Tsunami plugin. The knowledge base holds the information about an application, including interesting static files served by the application, unique hashes of these static files, and the release version of the hashes. Static files like logos and favicons will help the fingerprinter to identify the application as they are usually unique per software, while other files like scripts and stylesheets will help locate the versions as they are frequently updated across versions. To identify an unknown application we first crawl the application and collect all static files from the responses. Then we match their hashes against the knowledge base to identify the application and version. Consequently, this stage is by far the most time- and resource-intensive one, as crawling and downloading all embedded resources can result in several hundreds of requests for a single target.

Related Work The two most popular tools to conduct fast port scans are *masscan* [93] and *ZMap* [63], which were both released in 2013 and resulted in many publications making use of them in the following years. Similarly notable are the search engines *Shodan* [225] and *Censys* [62], which conduct regular scans and host a website that allows others to query their data, supporting researchers without the means to conduct scans themselves. However, checking for open ports and subsequently for Web services that respond to HTTP(S) requests is just the first part of our scanning pipeline. In particular, we also want to identify whether the deployed application suffers from a missing authentication vulnerability, which makes it related to works on *black-box web security scanners* [e.g., 124, 61, 60, 199, 71]. Yet these tools mostly focus on more widely known vulnerabilities such as SQL injection and XSS. Our tool, on the other hand, tries to detect MAVs in web applications, which were, to the best of our knowledge, not yet studied by the academic community and are also often not yet considered by commercial scanning tools, as we demonstrate in our paper.

Moreover, the fingerprinting methodology we employ is related to tools like *WhatWeb* [110] and *BlindElephant* [252]. WhatWeb uses a list of manually curated signatures to detect specific applications, i.e., it can only extract version information if it is voluntarily disclosed by the application to anonymous visitors. BlindElephant, on the other hand, tries to request as many resources, like images and stylesheets, as possible and then maps their hashes to known versions of previously indexed applications. However, we found BlindElephant to be severely outdated and discontinued and therefore created our own fingerprinting pipeline which combines the advantages of both these approaches into one.

Contribution The author of this thesis contributed to this scanning pipeline by designing and implementing the prefilter, about one-third of all our Tsunami detection plugins, the "glue" code combining the three stages, as well as all post-processing and data analysis code. Moreover, he designed the employed fingerprinting methodology and created a prototype of it during a 3-month research internship at Google in 2019. Guoli Ma implemented the general Tsunami framework and the final fingerprinter plugin used in the study. Manuel Karl, Sebastian Lekies, and Guoli Ma implemented the remaining MAV detection plugins.

2.1.2 General Scanning Challenges

What we could learn from the example of the Tsunami scanning pipeline is that the approach with multiple stages that act as a filter for subsequent slower stages scales well, if the amount of targets is extremely large and we also expect most of them to be out of scope for a detailed scan. Beyond this, the pipeline also has many other advantages such as a manageable complexity due to modularization into multiple stages and further into individual plugins for the later stages. However, the aforementioned design also has a significant limitation: It only works for *known vulnerabilities*. This is still useful, either as a way to quickly scan for affected hosts once the details of a particular vulnerability go public or as part of regression tests to make sure no further outdated versions are accidentally introduced to a network. However, to prevent vulnerabilities in custom applications under active development, e.g., a company's own web applications, a scanning pipeline would also need to detect whole *vulnerability classes* such as XSS. Moreover, if the scanner could identify generic vulnerabilities without having seen that particular instance before, that would also remove the burden of implementing a new detection plugin for every newly discovered vulnerability.

Back in 2006, Kals et al. [124] had already worked on trying to improve the state of the art for web scanners and presented their generic black-box vulnerability scanner called *Secubat*. Compared to previous works, it did not rely on a list of known vulnerability instances and could detect more than a single class; in this case both SQL injections and XSS flaws. This requirement of detecting vulnerability classes instead of specific vulnerabilities instances resulted in two major challenges many researchers attempted to tackle over the subsequent years, namely *crawling* and *vulnerability analysis*, which we will introduce in the following.

The crawling challenge When we want to detect whole vulnerability classes, we do not have the a priori knowledge anymore where exactly a vulnerability resides. So the question is now: How can we make sure to fully traverse the web application and interact with all discovered features, in order to execute as much client- and server-side code as possible? This is step is crucial, as only the functionality we discover during the crawling phase can later be investigated by the vulnerability analysis components of the scanner. If we already miss something here, we surely miss it in all later stages of the pipeline as well.

Notably, there is a big difference between the crawling that search engines like Google conduct for content indexing and crawling in the security context. The main reason for this is that search engines want to discover all *content* while security scanners want to discover all *functionality*. For example, given a blog with 100 posts on different subpages, a search engine would want to visit all these subpages to index their text content so that they can be included in its results. However, if only the text content differs on these 100 subpages, then from the perspective of a security scanner these are all identical, and visiting one would be enough, as they include the same functionality and execute exactly the same server-side logic. On the other hand, given a login subpage, the search engine will discover almost nothing interesting there, while the security scanner might send hundreds of requests to only this one page to test for weak credentials, authentication bypasses, or injection vulnerabilities. Therefore, previous research from the domain of search engines can not be directly reused for security scanning as the perspective and underlying requirements are very different. This is also the reason why we generally prefer the term *security scanning* throughout this thesis, which implies a crawling step but is optimized to discover *functionality over content*.

Back in 2010, Doupé et al. [61] conducted a comprehensive study on the limitations of existing black-box web vulnerability scanners in order to determine *why* they perform poorly. For this, they created *WackoPicko*, a vulnerable web application that they used as a testbed to evaluate eleven popular scanning tools. Their site contains several crawling challenges such as the use of HTML frames and malformed HTML in general, vulnerabilities hidden behind authentication or other multi-step actions, forms with file uploads, and an infinite calendar severing as a crawling trap. Moreover, they also evaluated the scanners' ability to extract links with 54 different ways of embedding links to other pages using WIVET [194]. They found that crawling is "as critical and challenging" as the vulnerability analysis step afterward, confirming that more research is needed to improve the discovery capabilities of security scanners. Additionally, incorrect parsing and lack of support for pervasive web technologies prevented the tools from reaching many of the vulnerable pages in the first place.

Application state Another specific limitation of the Tsunami pipeline that affects our crawling is that we were only looking for pre-authentication vulnerabilities, i.e., vulnerabilities exposed to all anonymous users. Obviously, this is a special case and a generic web vulnerability scanner would also need to support login functionality and deal with cookies to reach parts of the web application that are only available to authenticated users. In a more general sense, this means we have to consider the *state* of the web application, as visiting the same URL might lead to quite different results when logged in and when logged out. A typical example besides authentication is an e-commerce application where a user adds an item to their shopping cart and then proceeds to the checkout page. As long as the shopping cart is empty, the checkout page might either be unreachable or contain vastly different functionality than when we are in a state with items in our cart.

Dealing with application state is challenging, as black-box scanners can only observe the output of an application without knowledge of its internal state on the server-side. Specific challenges are *triggering state changes*, as we first need to find out which (potentially complex) actions cause a state change, *state navigation*, as URLs might not directly map to states, i.e., navigating back will not necessarily result in the previous state, and *state explosions*, as even simple applications can have infinite states [257]. In their publication called *Enemy of the State*, Doupé et al. [60] approached these challenges by trying to infer the internal state machine of the web application in a fully-automated fashion. They then use this partial model to crawl and fuzz the application in a state-aware manner. Depending on the complexity of the application under test, their evaluation showed that the state-aware scanner significantly increased the code coverage compared to simpler crawling tools like *wget* or *w3af* [212] that are not aware of application states.

The analysis challenge As previously discussed, the crawling functionality of a web security scanner is very important, since what we do not discover we also can not analyze. While the Tsunami pipeline could discover *known* vulnerabilities, the goal of a generic security scanner is much broader and such a scanner wants to detect whole vulnerability *classes* on the web pages discovered during the crawling process. The resulting analysis challenge basically boils down to the following question: Given a specific functionality of the website, how can we automatically determine if it is vulnerable or used in an attack? It should be noted that traditional security scanners usually are only interested in the former, i.e., *finding vulnerabilities*. However, in the context of this thesis, we will also use our security scanner to find ongoing attacks in the wild, i.e., websites that *distribute malicious code* to their visitors. Therefore, our scanner will share some similarities with features more often associated with anti-virus products such as VirusTotal [260] that visit user-submitted links and subsequently analyze that website.

Generally speaking, the analysis of websites can be divided into static and dynamic approaches. Since our scanner uses a black-box approach, it will always need a crawling component that dynamically explores the web application. So while we can not apply static analysis on the server-side code, static analysis of all discovered client-side code is still an option. However, the lack of types in JavaScript code as well the prevalent *minification*, i.e., renaming of variables and functions to shorter identifiers for less network traffic, means that the static analysis of JavaScript is rather challenging [227]. Therefore, dynamic analysis or a combination of static and dynamic analysis is often used instead when analyzing web applications for vulnerabilities and attacks.

The amount of works in the area of the analysis challenge is vast, so for brevity we will focus only on the analysis of one particular vulnerability class that will also become relevant in a later chapter, namely *Client-Side XSS*. While server-side XSS attacks had been known for a number of years, its client-side counterpart was only later discovered by Klein [132] in 2005 and happens if user-provided input is insecurely processed on the client-side instead. For example, the code in Figure 2.1 is meant to dynamically add an image to the DOM, which then sends both the query parameter as well as the URL fragment of the *including* site back to the advertisement company. However, this snippet suffers from a Client-Side XSS flaw, since both the query parameters and the fragment are simply concatenated with the HTML code — without any sanitization or encoding. It should be noted that the automatic encoding of parts of the URL is different between browsers and also has changed over time. In this thesis, we still consider the old behavior of not encoding the URL fragment in `location.hash`. Hence, an adversary can inject markup into the fragment of the URL to trigger the flaw, e.g., with the payload `'><script>alert(1)</script>` to demonstrate the attack with an alert box.

```
document.write("<img src='http://ad.com/ad.jpg?query=" + location.search + "&hash=" +
↪  location.hash + "'>");
```

Figure 2.1: Example of a Client-Side XSS vulnerability

From a conceptual standpoint, Client-Side XSS is caused when an unfiltered data flow occurs from an attacker-controlled source to a security-sensitive sink. In the case of such a Client-Side XSS, the source can be, e.g., the URL, whereas an example for a sink is `eval` or `document.write`. Therefore, a popular approach to detect these kinds of vulnerabilities is *taint tracking*, i.e., flagging all strings from those untrusted sources with "taint" and then propagating that taint information through the whole application to see if parts of it end up in one of the monitored sinks. For example, in 2013 Lekies et al. [144] created a modified version of the Chromium browser that supports taint tracking and discovered client-side XSS flaws on almost 10% of the Alexa top 5000 domains. For this, they not only needed a taint-aware JavaScript engine and DOM implementation but also a context-sensitive exploit generation to provide accurate analysis of the prevalence of these flaws. Since then, many works investigated this phenomenon further, such as the complexity [244] and history [241] of these vulnerabilities, a follow-up study on its prevalence in 2018 [156], an approach for the automatic discovery of a persistent variant [239], and improved techniques for exploit generation [20]. All these works have in common that they focus on improving the analysis techniques for just one specific vulnerability, thus showcasing the extent of the analysis challenge in a security scanner that likely wants to detect multiple of these complex vulnerability classes.

Emulated browsers The previously described challenges, as well as the complications caused by application states, mean that we need to bid goodbye to our highly performant but basic HTTP communication as used in the presented Tsunami pipeline in Section 2.1.1 and move towards something with more browser-like features like tolerant HTML-parsing, form submission and cookies support. As a potential solution, we could instead use so-called *emulated browsers* instead of plain HTTP libraries. While sometimes referred to as *simulated* browsers in literature, these implementations are actually much closer to an emulator, i.e., they only try to mimic the observable behavior of a real browser without running a full simulation of its internal state. Therefore, they should only be seen as rough approximations that do not support many features and edge cases. One of their advantages though is that they do not incur the high performance overhead of running a full desktop browser like *Google Chrome* or *Firefox*. Moreover, they are comparatively easy to integrate into a scanning pipeline, as they were designed to be controlled via a programmatic interface and not by an end-user with a graphical interface.

For example, *HTMLUnit* [27] describes itself as a "GUI-Less browser for Java programs" that was first released in 2002 and, at the time of writing, is still maintained almost 20 years later. It is mainly intended as a way to automate the testing of websites and can emulate Chrome, Firefox, or Internet Explorer. Among other features, HTMLUnit supports cookies, submitting forms, and provides wrappers for easy DOM access. Moreover, it uses the *Rhino engine*, which is also written in Java, to provide JavaScript support. HTMLUnit was used in several influential security papers, such as *Enemy of the State* [60], *You are what you include* [185], and *Revolver* [127]. Another popular emulated browser was *PhantomJS* [106], which launched in 2011. Compared to the custom Java implementation used by HTMLUnit, it is instead based on *WebKitQt*, i.e., a fork of the rendering engine *WebKit* used by Safari and earlier versions of Google Chrome. Therefore, PhantomJS behaves much closer to a real browser. It was used in various security and privacy publications, e.g., to detect website defacements [24], observe typosquatting [246], and audit third-party data collection [147]. However, unable to keep up with the fast evolution of browsers due to a lack of active contributors, the project was discontinued in 2018 [105].

2.2 Necessity of Modern Web Scanning

For now, it seems that emulated browsers might be a good solution to the scanning challenges introduced in the previous chapter. However, the Web continued to evolve ever more quickly: As one example, if we look back at the release history of the Firefox browser, we can also see that its development cycle speed up from approximately one major release per year around 2010 to a major release every month as of 2021 [168]. While the monthly releases most certainly do not contain as many features at once as the yearly releases might have, this is still a considerable change and means that new web standards result in changes for end-users much faster today. However, this also means it became much harder for the developers of emulated browsers to keep up with the constant introduction of new features to the Web platform.

This change of pace along with certain web development trends means that there are many reasons why even emulated browsers would not be good enough to comprehensively scan and analyzed modern websites today. In the context of this thesis, we will look at four web development trends in particular:

- **Complex client-side code**, i.e., high reliance on JavaScript support
- **Blurring of involved parties**, i.e., heavy use of third-party scripts
- **Volatile content**, i.e., websites that expose non-deterministic behavior
- **Use of emerging features** such as *WebWorkers*, *WebSockets* and *WebAssembly*

For each of them, we first introduce their technical details and then discuss how they affect web scanning endeavors and thus result in new research challenges. Moreover, for each trend, we also discuss noteworthy related works on that topic.

2.2.1 Complex Client-Side Code

Traditionally, the crawling component of a web security scanner found new URLs by extracting them from the HTML, e.g., by using regular expressions, but this is not sufficient anymore to explore modern websites [199]. The main reason for this is that they no longer consist of purely static HTML code, but instead part of the HTML might also be dynamically generated from client-side code. Because of this, elements that are important for a security scanner, such as forms and links, might only be present if the whole client-side code is executed and all resulting transformations are applied to the DOM. Figure 2.2 shows one such example that fetches further HTML content from a remote resource and adds it to the DOM after the initial page load.

```
window.onload = async function() {
    let response = await fetch("http://example.com/news/latest");
    document.querySelector("body > div").innerHTML = await response.text();
}
```

Figure 2.2: Dynamically adding further HTML content to the page

Back in 2012 already, researchers found that many URLs on modern websites were dynamically generated from client-side code [278]. In 2013, others found that almost all web applications had *hidden states* which can only be revealed after interacting with the website on the client-side and not be accessed directly via a specific URL [19]. Since then, the amount of client-side code only increased due to trends such as the introduction of *progressive web apps (PWAs)* in 2015, i.e., web applications that are designed to provide app-like features even when the user is currently offline [140]. Beyond the increased difficulties in traversing websites that make heavy use of JavaScript, this trend also leads to the involuntary introduction of entirely new vulnerabilities classes like client-side validation errors [221, 271] and DOM-XSS [132, 144], which is now usually called client-side XSS and will be the target of the protection mechanism presented in Chapter 3.

While emulated browsers generally have *some* JavaScript support due to its paramount importance on the Web, their support for modern features of the language is lacking. And while the *ECMAScript language specification* on which the JavaScript implementation is based had only two releases between 1999 and 2014, beginning with 2015 the specification received a major update on a yearly basis. In particular, the release in 2015, which is also known as ES2015 or ES6, added a vast amount of new syntax and features such as classes, modules, promises, proxies, and template literals [66]. However, as of 2021 the Rhino engine used by HTMLUnit still only supports a minority of the ES6 features, and almost no features which had been added with ECMAScript releases in the following years [171]. Therefore, a script that uses any of the new and popular features such as *async-await*, *let* and *const*, the *for-of* loop, or template literals would throw an error when executed with HTMLUnit.

Related work In the following, we briefly discuss a few publications that worked on improving scanners to make them function even on websites with lots of client-side code. In 2008 Mesbah et al. [157] presented *Crawljax*, which is a crawler with special support for AJAX websites. AJAX, which stands for *Asynchronous JavaScript and XML* and is based on asynchronous communication and DOM manipulations, was a popular way to create responsive sites back then. However, this new design paradigm presented a challenge for security scanners, as state changes do not necessarily have a corresponding URL anymore. In 2015, Pellegrino et al. [199] published their scanner called *jÄk*, which is combines traditional web crawling with dynamic program analysis. They found that their approach based on JavaScript API hooks and a runtime DOM analysis can explore 86% more of the surface of a web application than existing approaches, highlighting the relevance of this research challenge. Most recently, Eriksson et al. [71] published their work on *Black Widow*, a black-box approach to deep web crawling and scanning. In particular, they focus on combining approaches from previous works that provided solutions to individual crawling challenges such as asynchronous requests and inferring server-side state into a single scanner. This resulted in a further improved code coverage during scans when compared to previously discussed scanners such as *jÄk* and *Enemy of the State*.

2.2.2 Blurring of Involved Parties

Modern web applications consist of functionality originating from many different parties. To allow for easy integration into existing sites, such third-party functionality is often added via client-side JavaScript code. This is enabled by the fact that sites may include external JavaScript from other origins, which is subsequently executed in the origin of the *including* site. Hence, such code runs with full privileges, e.g., can modify the DOM to add a frame pointing to an advertisement or observe user interaction for analytic purposes. Apart from ads and analytics, other use cases for third-party code include location services, social media integration, or support functionality.

```
<head> <!-- HTML code delivered from https://shop.example -->
<script src="https://chat.example/service.js"></script>
</head>
[...]
<script> //Code of https://chat.example/service.js
if(!window.jQuery) {
   let script = document.createElement('script');
   script.type = "text/javascript";
   script.src = "https://code.jquery.com/jquery-latest.min.js";
   document.getElementsByTagName('head')[0].appendChild(script);
} </script>
```

Figure 2.3: Two different ways to include third-party scripts

The example in Figure 2.3 shows an online shop hosted at *shop.example* that wants to integrate a chat service provided by *chat.example*. The chat service, however, has a dependency to the popular *jQuery* library and does not want to bundle it with its own chat script to reduce network load. Therefore, they dynamically add the required script from *jQuery.com*. While reducing network load and increasing cache hits, this approach also has the advantage that it allows seamless updates, since changes to the third-party code are immediately "applied" to all including sites. However, this practice also has some severe consequences for the security of the first-party website. As all these scripts run in the same context, we are putting full trust into the servers that provide these scripts to us. If they get hacked or decide to turn rogue, they can directly deliver malicious client-side code into our page. Sometimes the attack vector might even be simpler, for example in 2012 Nikiforakis et al. [185] investigated inclusion relations on the top 10,000 sites and found various inclusions to IPs and domain names that were not allocated anymore, which thus could be obtained by an attacker.

In a more general sense, it is crucial to differentiate between actions of first- and third-party code. The user usually only has a trust relationship with the first party, e.g., if we order something from an online shop we obviously need to provide our postal address to that shop. However, if the third-party code is embedded directly into the first-party context, i.e., without any isolation through iframes, then these scripts also have access to all the sensitive data that we trusted the first-party with. In 2017, Acar et al. [1] investigated third-party scripts that exfiltrate sensitive data on a large scale. They found invasive practices such as inserting invisible login forms to trigger the browser's autofill feature, as well as third-party session replay libraries that exfiltrate the whole DOM including all personal information such as health conditions or financial data. Unfortunately, keeping track of which actions are conducted by which party is actually a fairly difficult task. For example, a first-party script might use code provided by the third-party library jQuery to insert further script code to the page, blurring the lines between the different parties. Correctly attributing actions to a certain party requires mechanisms that currently neither emulated nor real browsers provide out of the box.

Related work In the following, we briefly summarize a few recent works on the topic of third-party script inclusions. In 2017, Kumar et al. [137] focused on the structure of these inclusions and introduced the concept of implicit trust. They furthermore showed that a quarter of the top 1 million sites were blocked from deploying HTTPS due to their inclusions. The risks of including outdated third-party libraries were analyzed by Lauinger et al. [138], showing that 37% of the top 75,000 sites include at least one library containing a vulnerability. In 2019, Ikram et al. [114] investigated how often malicious inclusions happen over implicit trust relations in the Alexa top 200,000. They found that 73% of the websites in their study loaded one or more scripts from third parties labeled as suspicious by VirusTotal. With respect to detecting third-party hosting, Matic et al. [150] proposed to use RDAP information about the resolved IPs of sites as well as information extracted from the start pages to detect hosting environments. In particular, they investigate if a given site is self-hosted or provided via a CDN or third party.

2.2.3 Volatile Content

While the previous trends motivated the use of a real browser to correctly deal with modern websites, there are cases where even that is not enough. Some analyses might require multiple visits to the same URL from slightly different environments, e.g., to confirm the presence of a vulnerability by comparing different execution traces. However, websites can be highly *volatile* and multiple visits even within a few seconds might yield different responses. One typical example is a front page like `reddit.com` that shows the most recent or most popular posts and constantly changes. In this case, only the content might change while still executing the same code on each page load, but another reason for volatile pages are ads and ad bidding in particular, where the actual ad is determined in real-time and might behave differently on each page load. One recent work that highlights the extent of volatility is the study by Urban et al. [254] from 2020, in which they create trees that represent the relationships between third-party script includes. They found that the third parties were often non-deterministic, with around 50% of the branches in their tree changing between repeated visits of the same page.

The previously described reasons make the analysis of benign websites unintentionally harder, as no two visits might execute exactly the same code. However, when investigating web attacks in the wild, attackers also can make it *intentionally hard* to reproduce and analyze incidents later, thus requiring a fine-grained recording of all browsing activities. For example, beyond very short-lived attacks in general, the payload might be only delivered to very specific victim environments, might require specific user interactions to trigger, and might be delivered via malicious advertisements [146]. As popular browsers such as Chrome and Firefox do not provide a fine-grained recording out of the box, there were multiple works that attempted to create a comprehensive replaying system with acceptable overhead, as we will discuss in the following.

Related work On one hand, several so-called *record-and-replay* (R&R) systems have been proposed over the years, such as *Webcapsule* [182] which is an instrumented version of Google Blink's rendering engine and the corresponding V8 JavaScript engine. However, it does not collect JS-level events that are needed for a deterministic replay, prompting the development of *JSgraph* [146] which features even more extensive recording capabilities. Compared to recording and replaying whole virtual machines of all system-level events, these browser modifications are rather lightweight. On the other hand, they still introduce a considerable overhead and resulted in works that instead enhance the browser's logging capabilities, such as *ChromePic* [255] and *Mnemosyne* [7]. Compared to ChromePic, Mnemosyne is far more portable, as it does not require any browser modifications and is instead deployed via a passive auditing demon. The main advantage of these approaches is the lower performance overhead and smaller storage requirements compared to a full R&R system but this naturally comes at the cost of completeness, as even the extended logs are not guaranteed to contain all necessary data for a successful reconstruction of an attack.

2.2.4 Use of Emerging Features

Previously, we discussed the general trend of using more and more JavaScript on the Web and the many additions to the core language over the recent years, such as new keywords. In addition to that, there is also the separate trend of browser vendors continuing to add more features in the form of APIs accessible from JavaScript code. As one example of such an API, take another look at the code example in Figure 2.2. In this case, `fetch()` is short for `window.fetch()`, which is an API provided by the browser to conduct HTTP requests via JavaScript code. The fetch-API serves as a more powerful replacement for the old but popular `XMLHttpRequest` API. Both these APIs are not part of the JavaScript language specification and are not supported when running JavaScript outside of a browser environment. While Node.js has similar functionality in the `request` module, many other APIs have no equivalent outside of the browser, e.g., because they interact with the DOM.

As an approximation for the increasing number of features, we can look at the number of properties that were part of the global `window` object over time. Based on the MDN compatibility data [169], we found that this number increased from about 250 properties in Chrome version 4 at the beginning of 2010 to over 900 about 11 years later in Chrome version 88 as Fig. 2.4 shows. This is only a very rough approximation as properties were not resolved recursively, e.g., everything below of `window.document` was not counted in this case, so the actual number is much higher. However, the comparison over time nevertheless serves to highlight just how many new features get added to the browser each year.

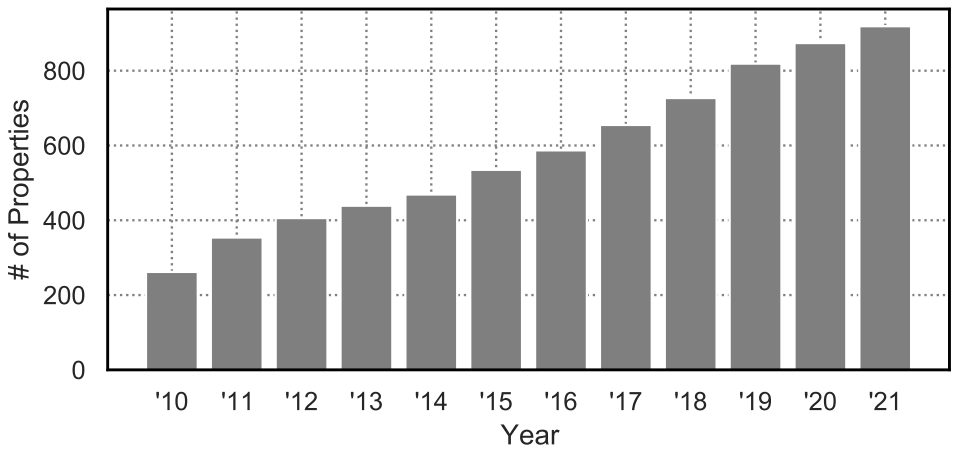

Figure 2.4: Number of properties of the global window object in Chrome over time.

In the following, we focus on three particular features because (a) their prevalence has been rising over the recent years, (b) they would be hard to support in an emulated browser without significant engineering efforts, and (c) they are useful from an attacker's perspective, as we will describe in the following with examples from related work, as well as in more detail with one particular attack that combines all of these three technologies in Chapter 5.

WebSockets The *WebSocket* protocol has been standardized as additional browser functionality in 2011 [77] and enables full-duplex communication from the browser to a web server with less overhead than HTTP. In particular, WebSockets are useful for web applications that need updates from the server multiple times per second, such as multiplayer action games and collaboration services, where multiple users write in the same document. According to the anonymous JavaScript feature usage stats collected by Chrome, WebSockets were used on about 9% of all page loads in 2021, and this number has slowly but steadily increased over the last few years [34]. It should be noted that these statistics are biased due to relatively few popular pages creating most of the page loads. For comparison, a study [174] conducted in 2020 found only 5.5% of the top 1 million websites to be using WebSockets, while the Chrome stats reported a prevalence of 8.5% on all *pages loads* at the same point in time. Moreover, the study also found that an overwhelming 95% of all scripts that initiate WebSocket connections are served by third parties, suggesting that this technology is, so far, mostly used for tracking, analytics, and advertisements. One noteworthy example of how such an emerging feature can be abused was reported by Bashir et al. [15] in 2018 where they describe how advertisement and analytics companies were abusing a bug in Chrome's `webRequest`-API that prevented ad-block extensions to block their WebSocket traffic.

WebWorkers Another addition was the so-called *WebWorkers*, which have been introduced in 2015 [104]. This programming primitive enables JavaScript code to schedule multiple threads and conduct concurrent computations in the background. While the original programming model underlying JavaScript already supports event-driven concurrency, orchestrating the available computing resources, such as multiple cores, has been technically involved. This problem is alleviated with WebWorkers, where the number of concurrent threads can be scaled with the available processor cores easily. According to the Chrome usage stats, workers are used on about 12% of all page loads in 2021 and this number has almost doubled over the last three years [35]. However, this number does not only include WebWorkers but all types of workers such *ServiceWorkers* as well. One of the possible abuses of WebWorkers is for *thread spraying*, in which many threads are created to fill up the available memory space in order to circumvent low-level countermeasures such as *code-pointer integrity* [82].

WebAssembly While WebWorkers allow to use more cores at the same time, JavaScript code is still rather inefficient as it requires a costly parsing and interpretation within the browser. This problem was addressed by the *WebAssembly* standard from 2017 [217]. The standard proposes a low-level bytecode language that is a portable target for compilation of high-level languages, such as C/C++ and Rust. WebAssembly code, or Wasm code for short, is executed on a stack-based virtual machine in the browser and improves the execution as well as loading time over JavaScript code [99]. Compared to the other two presented technologies, its usage is still relatively low, with about 1.8% of all page loads at the end of 2021 up from 0.3% in 2019 [33]. Nevertheless, WebAssembly already has been actively abused by websites shortly after it became available in browsers. In particular, the performance gains enabled efficient *cryptojacking*, an attack that covertly abuses the computing power of all website visitors to mine for cryptocurrencies and will be the focus of Chapter 5.

To summarize, we have seen that browsers constantly evolve and add new APIs that are relevant for security research. This means a modern website will only be fully functional if we visit it with an user agent that supports emerging features such as WebSockets and WebWorkers. Thus, even if we had access to an emulated browser that uses a recent JavaScript engine, these emerging features would need to be implemented into their emulation as well – a most likely infeasible task for small development teams, especially for complex features such as WebAssembly.

2.3 Browser Instrumentation

The web development trends in the previous section highlighted the need for a real browser over an emulated solution. This way, our security scanner can keep up with the constant changes of the web platform, as we will always support the latest features as long as these browsers get regular updates. However, using a real browser also results in new challenges, as we will discuss here.

One drawback is that running a full browser comes with a serious performance impact. For one, parsing and rendering HTML code and then additionally executing JavaScript code is a CPU-heavy process. Moreover, modern browsers are split into many processes, each of which might consume a considerable amount of memory. Last but not least, because a browser will require all embedded resources such as scripts, stylesheets, and images, the amount of network traffic required to load a page also significantly increases. However, as in most cases we are not interested in how the web page would actually look like for a visitor, we can apply some optimizations. For example, we could replace all images and videos with a locally cached placeholder to reduce the network load. Going further, we can even try to disable all parts of the browser that we do not need for an automated scanner, such as the whole browser GUI. Fortunately, browser vendors also recognized this need and recently began to add command-line switches which achieve exactly that. For example, Chrome released *Headless Chrome* [21] in 2017, which is optimized for running in server environments and reduces the CPU and RAM overhead by disabling features that are unnecessary for that use case.

Another drawback is that the massive size of the codebase results in a much larger attack surface for real browsers compared to tools with a very narrow scope like `curl`. As we are willingly exposing this attack surface to any website that we visit during our crawls, we must diligently keep the underlying browser up to date to avoid compromise through publicly disclosed exploits. In Section 6 we will discuss this security threat in more detail and present a study on the prevalence of outdated browsers running on the *server-side*.

However, the main challenge that we will discuss in this section, is that these desktop browsers were traditionally not designed with automation in mind, especially before the headless feature had been added. Therefore, we need *browser instrumentation*, i.e., we need a way to programmatically interact with a real browser to control it and extract the data that we are looking for in our experiments. In this section, we will first briefly describe a few useful approaches on how such an instrumentation can be implemented and then give a concrete, real-life example that was used in one of our publications.

2.3.1 Overview of Approaches

Generally speaking, our browser instrumentation needs to do two things: 1) control the browser, e.g., open a new tab and navigate it to a website, wait a certain amount of time, handle errors and crashes, etc. and 2) record and extract the information related to a vulnerability or an attack that our scanner is actually looking for. In the following, we will introduce several different instrumentation approaches that all have their unique advantages and drawbacks.

JavaScript wrappers First, let us focus on the second step, which can often be solved via JavaScript instrumentation using only features of the language and thus independent of any browser. For example, suppose we want to monitor all calls to a certain JavaScript function, e.g.,`document.write` because if used insecurely the function can lead to injection vulnerabilities such as client-side XSS that our scanner wants to detect. One way to achieve this is through *monkey patching*, i.e., wrapping the function to extend its functionality as shown in Figure 2.5. As long as we would execute these wrappers before any other code on the website can save a reference to the original function, all calls would be logged thanks to our JavaScript instrumentation.

```
let original = document.write;
document.write = function() {
    // Could report the call and parameters to our backend here
    logCall(arguments);
    // Call the original function
    original.call(document, ...arguments);
};
```

Figure 2.5: Monkey patching a JavaScript function

Creating wrappers for security-relevant JavaScript functions is a proven concept in web security research and was used in publications for more than one decade. For example, it allows the creation of lightweight security mechanisms [201, 248] without the need for browser modifications. Instead of merely logging the calls as shown in Figure 2.5, the parameters can also be checked against a policy. If the supplied values do not adhere to a predefined policy, the call to the original function is not executed, preventing access to protected functionality. By writing the protection mechanism entirely in JavaScript, it is easy to deploy by bundling it with the website itself and also does not incur the cost of maintaining as the browser's codebase changes over the years. In Section 3, we will present one protection mechanism against client-side XSS in detail, which is implemented entirely using such a JavaScript instrumentation.

Network interception While the JavaScript wrappers solve the problem of recording the relevant data – at least as long as the data is related to the usage of JavaScript features – we still need a way to automatically inject these wrappers into all websites that we load with our scanner. As previously outlined, it is important to ensure that our wrappers are executed before any other code on the page. Again, we first take a look at one generic solution by approaching this on the network level. By using a local proxy server to act as a man-in-the-middle for the scanner, we could inspect and modify all responses from the server. The main advantage is that the proxy is completely independent of the underlying browser, therefore we do not need to keep up with upstream changes in the browser. However, it also has severe disadvantages that make this solution generally undesirable.

First of all, we need to either support all the various content and transfer encodings to actually inspect the traffic or need to strip all unsupported ones from the outgoing request headers. Then, we need to parse the responses looking for HTML documents in order to add our instrumentation script to the very top as an inline script that will be executed first. However, security headers such as CSP might disallow our modifications and in particular any inline scripts, which is why we would need to rewrite these headers as well, which might later cause false positives during analysis due to the weakened settings. Even if we do account for all of this, there is still the problem that one document can create many different environments that would not be instrumented. For example, any dynamically created iframes or WebWorkers run in a different global context and thus would be missing our instrumentation that is running only in the main `window`. While one might attempt to also wrap the creation of all iframes, workers, etc. so that they propagate the instrumentation, it is very complicated to implement this in a correct and complete way. Therefore, network interception outside of the browser can not be recommended for instrumentation purposes.

Browser APIs Instead of trying to solve everything outside of the browser, we could also use well-defined APIs offered by the browser. The main advantage of this approach is that the browser does all the heavy lifting, e.g., decoding and parsing all responses if we want to intercept network requests. Moreover, we often do not low-level access to requests anymore as we could just instruct the browser to execute our JavaScript instrumentation code every time a new execution context is created. One option for this comes in the form of *browser extensions*, which are small modules written in HTML, CSS, and JavaScript that can extend the core functionality of the browser. The code for each extension runs in a separate execution context, this means if we were to overwrite built-in functions with our wrappers, the wrappers would only be active for code that runs *inside* our extension. However, assuming the extension has the correct permissions, it can inject code into the execution context of the loaded website as shown in Figure 2.6.

```
let script = document.createElement('script');
script.textContent = `/* Add the code to inject here*/`;
document.head.appendChild(script);
```

Figure 2.6: Injecting code from an extension into the document

While these extensions are portable and provide a mostly stable and somewhat browser-independent API, they also are the least powerful option for browser instrumentation purposes. For example, due to Chrome's architecture the often requested feature of modifying the response body of intercepted requests is, as of 2021, still not available with also no plans to add this feature in the future [45] Moreover, browser vendors are increasingly locking down the capabilities of extensions due to wide-spread abuse, such as unwanted ad injections [251]. An alternative to creating browser instrumentation via an extension is to instead use the *WebDriver API* offered by the Selenium testing framework [223]. As this browser instrumentation relies on the installation of additional native programs to interact with the browser, it can circumvent some of the limitations of browser extensions. Compared to the previously outlined approaches, the WebDriver API was specifically designed with browser instrumentation in mind. Figure 2.7 shows a complete example that uses a real browser to load the given URL, waits until the document has fully loaded and then searches for a specific element in the DOM. However, its API is still rather limited compared to the alternative that we will discuss next.

```
const documentInitialised = () => driver.executeScript("return initialised");
await driver.get("http://example.com");
await driver.wait(() => documentInitialised(), 10000);
const element = driver.findElement(By.css("p"));
```

Figure 2.7: How to control the browser via the WebDriver API

Chrome DevTools Protocol (CDP) While browser extensions and the WebDriver were a good work-around for the lack of an instrumentation API in the past, modern browsers nowadays ship with a much more powerful interface by default. This interface is based on the integrated *Developer Tools*, or *DevTools* for short. The DevTools is a GUI that acts like an IDE for web development inside the browser that can be attached to any tab. For example, the DevTools allow to inspect and modify the DOM and ship with a debugger for client-side code that supports breakpoints and stepping through the code. Moreover, the DevTools also allow the execution of JavaScript in an interactive shell and offer a way to inspect all network traffic including response bodies as well as headers and timing data. On top of that, the DevTools offer many advanced features like measuring site performance with a stack-based profiler, creating a heap snapshot to investigate memory leaks, and the ability to measure and inspect code coverage.

All functionally that the DevTools offer is also accessible programmatically through the *Chrome DevTools Protocol* (CDP) [38] for all browsers based on Chrome, while Firefox offers somewhat similar functionality with their *Remote Protocol* [78], which implements a subset of the CDP. Internally, the CDP uses serialized JSON objects and is divided into *domains*, each of which offers many commands and events that can be subscribed to. To enable a more convenient usage of the protocol, wrappers for many programming languages such as Node.js and Python exist. For example, Figure 2.8 shows how to control an instance of Chrome from Node.js via the CDP, using the *chrome-remote-interface* library [58].

In the following, we will introduce a few of the most useful features that the CDP offers for browser instrumentation: First of all, the *page* domain can be used to navigate to

```
//Launch the browser (with optional command line flags)
const port = await browser.initialize(flags);
//Connect to the CDP via the debugging port, open a new tab and enable the page domain
const target = await CDP.New({port: port});
const client = await CDP({target: target});
await client.Page.enable();
//Navigate to a website and wait for it to load
await client.Page.navigate({url: "https://tu-braunschweig.de"});
await client.Page.loadEventFired();
```

Figure 2.8: How to control a browser via the CDP from Node.js code

a URL, capture screenshots, and manipulate the navigation history. Moreover, it allows defining JavaScript code that is executed in every frame upon creation before the frame's scripts, including the main frame of the website during the initial navigation. The *debugger* domain, on the other hand, emits an event every time JavaScript code is parsed, which includes dynamically created code like via `eval` or the `Function` constructor. Other useful features of the debugger domain are setting conditional breakpoints and evaluating code in any scope. As the name suggests, the *network* domain offers useful features related to recording and intercepting network requests. Additionally, it allows to manipulate cookies, to define additional HTTP headers for all requests, as well as to change the default user agent string.

Compared to browser extensions and Selenium's WebDriver, the API of the CDP is less stable with many methods marked as *experimental* that might change in future versions of Chrome without advance notice. On the other hand, the CDP enables analyses that would be impossible on the network level or via a mere extension, such as running a code profiler that records which JavaScript statements were executed for precise code coverage information. To make the CDP even more accessible than the low-level wrappers as shown above, Google built the *Puppeteer* [85] library for Node.js which provides a more high-level API but internally uses the CDP. As each Puppeteer version is bundled with one specific version of Chrome, it is also guaranteed to work even if the underlying unstable CDP changes over time.

Native instrumentation While the CDP is certainly a powerful option for browser instrumentation, it does not allow *arbitrary* changes to the behavior of the browser and its JavaScript execution. In some cases, it might be necessary to modify the browser's source code to add the required logging capabilities or functionality changes directly, resulting in *native instrumentation*. Obviously, this allows for complete control over the browser, but finding the correct location and then implementing a working change can represent a challenge for everyone who is not familiar with the codebase. For instance, in 2020 the Chromium browser had over 25M lines of code, of which around half are written in C++ [23]. Moreover, another challenge is to constantly maintain these changes as new browser versions are getting released which can result in complex merge conflicts that need to be manually resolved. This means that even if the authors make their browser patches available after the publication of a project based on native instrumentation, this does not guarantee that the patches will remain useful in the future. Overall, native instrumentation by patching the browser should only be attempted as a last resort if the attempted analysis can not be realized through other means such as the CDP. One example of a use case that often only can be solved via native instrumentation is the forensic record and replay engines like *WebCapsule* [182] that were discussed in Section 2.2.3.

2.3.2 Exemplary Instrumentation: Smurf

In Section 2.2 we described the web development trend of heavily relying on third party scripts, which results in a blurring of the involved parties and their code. This analysis challenge will now serve as a real-life example of how such challenges can be solved by instrumenting a real browser with the CDP.

In our paper "Who's Hosting the Block Party?" [238], we set out to understand to what extent the trend of outsourcing functionality to third parties has adverse effects on two key security mechanisms: *Content Security Policy* (CSP) and *Subresource Integrity* (SRI). CSP primarily aims to mitigate the impact of XSS vulnerabilities [236] while SRI aims to secure including sites against compromise of third-party servers by only executing scripts that match the cryptographic hash attached to their definition [172]. Unfortunately, both mechanisms lack widespread deployment [10]. Assuming that a first-party wants to deploy CSP and SRI and is able to make their codebase compliant with these mechanisms, we assess how many sites could fully deploy the mechanisms without cooperation from their third parties. To enable these detailed analyses, we needed to accurately depict trust relations to create holistic views into inclusion chains within all pages of the investigated sites. In what follows, we describe how our instrumentation can keep track of the code's *provenance* even for dynamically inserted scripts, showcasing the advantages of using the CDP.

Script inclusion terminology Before going into the details of our analysis, we first provide a brief recapitulation on script inclusions and introduce our terminology. On the initial delivery of an HTML document to the browser, developers can include the code inline into the document, as well as point the browser to the URL from which an additional script should be fetched. Any included script resource can interact with the full DOM functionality of the web application, which allows inclusions to display ads or augment native DOM functionality. As outlined in Section 2.2, this essentially enables all running code to further conduct additional inclusions via the addition of dynamic script. For example, this can be achieved by writing script tags or event handlers through `document.write`, invoking `eval` to convert a string to code, or programmatically adding scripts to the DOM through `document.createElement` and `appendChild`. By default, inclusions cannot be restricted, i.e., any included script can add additional content to its liking. Therefore, our instrumentation needs to handle all these different variants that can be used to include scripts to keep track of their *initiator*, i.e., the original script that caused the new inclusion to happen. By connecting all nested inclusions with their respective initiators, we obtain an *inclusion tree* for each web page.

Precisely Capturing Inclusion Relations To analyze inclusion relations, we first need to record which entity initiated a particular script inclusion. For this, we rely on the stack traces that the CDP provides for most events. Events such as `Network.requestWillBeSent` allow us to register a callback that can log the stack traces for dynamically inserted script tags with a URL in the `src` attribute. The `Debugger.scriptParsed`, on the other hand, provides stack traces for dynamically created via `eval` and similar functions. Figure 2.9 demonstrates how simple it is to save the stack traces of these events in a hashtable to resolve them later.

```
let networkTraces = new Map();
client.Network.requestWillBeSent(function(params) {
    if (params.type == "Script" && params.initiator && params.initiator.stack) {
        networkTraces.set(params.request.url, params.initiator.stack);
}});

let debuggerTraces = new Map();
client.Debugger.scriptParsed(async function(params) {
    if (params.stackTrace) {
        debuggerTraces.set(params.scriptId, params.stackTrace);
}});
```

Figure 2.9: Saving stack traces of dynamically inserted scripts in a hash table. The `params` are values that the CDP provides to these callbacks.

For inline scripts and event handlers, we need to rely on additional JavaScript instrumentation for DOM manipulations such as `document.write` and `Element.innerHTML` to get accurate stack traces. Our custom JavaScript wrappers then notify our Node.js backend, every time an inline script or event handler is dynamically inserted and enable us to save the stack trace in these cases as well. Figure 2.10 shows how such a wrapper would work in detail at the example of event handlers. This instrumentation only works because the CDP attaches stack traces to `console.log` calls. Moreover, it relies on the `Page.addScriptToEvaluateOnNewDocument` function to create these wrappers in all windows and iframes.

Resolving stack traces As we have seen, the CDP makes capturing the stack traces for most variants of script inclusions relatively easy. Now, we need to traverse these stack traces to identify the initiator of the respective inclusion. What makes this a bit more difficult is that because of modern JavaScript features such as Promises, call stacks can be *asynchronous*. And while async stack traces have been enabled by default in Chrome since

```
// First, use monkey patching to wrap all functions like document.write to call
// this function first, then the original (not shown for brevity)
function intercept(htmlCode) {
    let fakeDOM = (new DomParser()).parseFromString(htmlCode, "text/html");
    // Dealing with event handlers
    for (let ele of fakeDOM.getElementsByTagName("*")) {
        for (let attr of ele.getAttributeNames()) {
            if (attr.startsWith("on")) {
                // Create a short identifier for the event handler
                let hash = hashCode(ele.getAttribute(attr));
                // Notify our backend to attach the stack trace caused by this log event
                // to the identifier that we include in the message
                console.log("[Initiator found] " + hash);
            }
        }
    }
}
```

Figure 2.10: Creating stack traces for dynamically inserted event handlers via JavaScript wrappers

2017 [16], these cases still require special consideration in the implementation. In particular, all preceding stack traces must be recursively resolved via the `Debugger.getStackTrace` API of the CDP with the current trace's `parentId` in the arguments to get the *full* chain of events that led to a given inclusion. With the increasing language support, e.g. `async/await` in ECMAScript 2017 [67], we believe not handling these cases correctly to be an emerging threat to such analyses. So while the actual implementation details are not needed to understand the rest of this theses, they serve as a practical example of how working with the CDP can result in more accurate results.

Contribution In the spirit of open science, the implementation was open-sourced and is available as part of a lightweight analysis tool called *SMURF Monitor Unveils Roadblocking Features* [81]. SMURF is designed to help developers uncovering potentially dangerous inclusion decisions and allows other researchers to compare against our work. The author of this thesis contributed the mechanisms described in this section, which are capable of collecting precise inclusions relations that were used for the data collection in our paper, as well as in the open-source counterpart. The remaining implementation as well as all analyses on the collected data were conducted by Marius Steffens, except the analysis on code drift which was conducted by Ben Stock. Therefore, further details about SMURF and our results on CSP and SRI blockage in the wild were omitted from this thesis and can be found in the resulting publication [238].

2.4 Summary

In this chapter, we first took a deeper look at a security scanning pipeline using the example of Tsunami, which had been designed to find a predefined list of pre-authentication vulnerabilities in web applications. From there, we discussed how to extend such a pipeline so that we can find unknown instances of known vulnerability classes. However, this transition resulted in new crawling and analysis challenges and we saw emulated browsers as one potential improvement for the pipeline. Then, we investigated four web development trends in detail and found that rather than emulated browsers, we need to rely on real browsers to keep up with the constant evolution of the web platform. Thus, we examined several browser instrumentation approaches that turn the browser into a crawling and analysis tool that can be integrated into our pipeline. Finally, we concluded with a showcase of how browser instrumentation can solve one of the analysis challenges using the example of SMURF.

Now that we have established browser instrumentation as a useful tool for security scanning, we describe three use cases in more detail in the following. First, in Section 3 we create a mechanism to measure the compatibility of a novel defense for client-side XSS, which is complicated by the analysis challenge of blurred third-party code. Then, in Section 4 we explore abuses of JavaScript capabilities that allow malicious websites to detect if they are under analysis, where we ran into the crawling challenge of volatile websites. After that, in Section 5 we investigate the malicious usage of WebAssembly for parasitic computing and obfuscation, which is an analysis challenge due to their reliance on many emerging technologies. The trend of heavy usage of JavaScript affects all three chapters and is one of the main motivations to conduct these experiments in a real browser in the first place. However, in Section 6 we present a study on the usage of instrumented browsers in the wild and highlight how they can, unfortunately, also introduce new vulnerabilities.

3 Measuring Compatibility of an XSS Defense

In Section 2.1.2 we saw how the insecure usage of attacker-controllable data in functions like `eval` or `document.write` may cause a Client-Side XSS vulnerability. On the other hand, in Section 2.2 we outlined that web applications nowadays often contain code from many different parties and that this third-party code runs with full privileges in the origin of the *including* site. To prevent the accidental introduction of Client-Side XSS vulnerabilities into a otherwise secure website through third-party code, we designed a protection mechanism called *ScriptProtect* [177]. In this chapter, we will describe how a modern security scanner based on a headless browser can be used to evaluate the compatibility of this XSS defense. In particular, this chapter will focus on the analysis challenge of *blurred third-party code* as a motivation for our scanning methodology. Moreover, this chapter serves as an interesting use-case because we are not only looking for security flaws but simultaneously also test if applying our novel protection mechanism would have prevented the exploitation without breaking any functionality of the website.

In Section 3.1, we first motivate the problem and briefly introduce ScriptProtect's approach. Then, we outline our threat model and explain what is in and out of scope for the protection mechanism. Next, in Section 3.2 we describe the two different protection modes that ScriptProtect offers to support both existing and new web applications, which are based on a dynamic access control and the introduction of unsafe variants, respectively. Moreover, we also introduce all dangerous APIs that are covered by ScriptProtect in the same section. After that, we explain our methodology based on the instrumentation of these dangerous APIs in Section 3.3, with an emphasis on the different implementation approaches for the different types of APIs. In Section 3.4, we then follow up with an evaluation of our proposed defense in a real-world scenario, where we examine the compatibility, effectiveness, and performance of ScriptProtect in detail. Finally, in Section 3.5 we discuss publications related to Client-Side XSS protections and the securing of third-party code.

3.1 Use Case: Third-Party Client-Side XSS

The direct client-side inclusion of cross-origin JavaScript resources in web applications is a pervasive practice to consume third-party services and to utilize externally provided libraries. The first-party code is under the control of the site's developer, who may employ secure coding practices to avoid vulnerabilities or use tools to find and patch them [196]. In contrast, the developer has no control over third-party code and cannot address vulnerabilities in included code. Specifically, if a third-party script is included, it has the power to add additional scripting content to the including site. Given this model of including scripts from other parties, flaws in third-party code result in vulnerabilities that directly affect the including website. As we will see in this chapter, a significant fraction of all Client-Side XSS flaws is caused by such third-party code. Thus, vulnerabilities are introduced in otherwise secure sites merely by the inclusion of such a benign-but-buggy third-party script.

3.1.1 ScriptProtect

A study from 2018 by Melicher et al. [156] has shown that the general threat of Client-Side XSS is still widespread in modern web applications. Moreover, as an earlier work from Stock et al. [244] has shown, a significant fraction of the exploitable flaws is caused by third-party content. This paints a grim picture for the security-aware developer who invests time in securing the web applications against the threat of XSS, only then to notice that a third party introduced a vulnerability. Especially given that modern web applications and their interconnectivity appear to grow even further [241], it is unreasonable to burden developers with banishing third-parties from their security perimeter. Instead, we would want an environment that prevents other parties from inadvertently introducing new vulnerabilities. This way, we enable first-party developers to focus on the security and functionality of their own code. More specifically, we want to allow the third party to add benign content like images, but ensure they cannot add markup containing script code such as event handlers. For this, we propose *ScriptProtect*, which functions as a lightweight drop-in solution to harden a web application against benign but buggy third-parties. At its core, ScriptProtect ensures that third-party code is unable to accidentally add unsafe markup into a document. Before we describe this protection mechanism in more detail, we first introduce the threat model we considered for its design.

3.1.2 Threat Model

Over the past decade, there have been many attempts to allow the inclusion of third-party code without compromising the security of the including site itself [5, 115, 201, 248]. However, these papers assume a *malicious* third-party and hence require very strict isolation of the third-party code, which in turn tends to break many use cases like analytics and enrichment of pages in general. In contrast, in the context of ScriptProtect, we consider the third-party scripts to be *non-malicious* but *vulnerable*. This means that the initially executed third-party code has no intent of undermining our protection scheme. While the trusted third party is meant to operate within the security bounds of the first-party application, a attacker can try to attack in three distinct ways. First, they can try to inject malicious inline scripts. Second, when an HTML injection is possible, they can resort to including an externally hosted JavaScript file containing their payload. Finally, they can use `eval` or similar functions to conduct a string-to-code transformation and gain code execution.

The attacker in this scenario is a malicious actor, who aims to exploit the Client-Side XSS problem, which was unwillingly introduced by the third-party script. As such, the attacker has to perform a string-to-code conversion while abusing an XSS flaw to introduce his new, malicious code into the origin. Hence, while a malicious third party could undermine our protection scheme, e.g., by getting a reference to an unprotected version of `document.write` from a newly created frame, the attacker in our scenario does not have that capability. The honest third party, on the other hand, was trusted through the action of including a script from its host. As such, a malicious third-party has no need to circumvent our protection, e.g. of `document.write`, as it can already execute arbitrary JavaScript code without requiring additional scripts elements in the DOM. Therefore, our goal is to prevent third parties from *accidentally* introducing new code where writing passive markup – without scripts and event handlers – to the page would be sufficient. Consequently, attacks that involve compromising the third-party servers are out of scope for our protection.

3.2 System Overview

On a conceptual level, ScriptProtect works as follows: All potentially dangerous browser APIs and properties which could cause Client-Side XSS vulnerabilities, e.g., `innerHTML` or `document.write`, are instrumented at runtime. This is achieved by our protection mechanism which is simply included by a developer in the `head` portion of the hosting HTML document as an external script resource before any other external script is loaded. After the execution of ScriptProtect's code, all instrumented APIs are *secure-by-default*, meaning they cannot be used to add additional script content to the DOM.

3.2.1 Protection Modes

To allow trusted first-party code to still use the script-introducing functionality of these APIs, we offer two different protection modes:

New applications — Unsafe API variants While the standard API remains unable to introduce new script content into the web document, ScriptProtect introduces a second version of the API, e.g., `document.unsafeWrite`, to be used explicitly by first-party code in cases in which *additional script code should be added* to the document. As these cases are in the minority and clearly marked through the explicit usage of the unsafe versions, it is straight forward to audit such occurrences during development to avoid vulnerabilities. On the other hand, third-party scripts won't use the unsafe variants, as they are designed and implemented for standard browser functionality, and thus, are completely safe. If third parties are aware of the unsafe variant, they could obviously simply call that. However, as discussed in our threat model, we assume no intentional circumvention of our protection from a party that already has achieved code execution anyways.

Using these unsafe API variants (if really necessary) is the recommended way in case a new web application is created from scratch. However, as this change merely renames the original functionality, this can also be used for existing applications in combination with a rewrite proxy or static analysis tool. While giving the first party full access to the dangerous APIs by default is not ideal, we see this option useful for a transitional phase: access for the third party is immediately blocked while the first-party code is rewritten to use the unsafe variant by default. Then gradually all usages of the dangerous APIs need to reviewed and, depending on each individual case, changed to use the safe variant, if possible.

Existing applications — Dynamic access control In case a significant codebase for the target application already exists, ScriptProtect offers the option to dynamically adapt the behavior of the APIs and DOM properties, depending on the calling party. In this case, the mechanism transparently checks the stack trace of the current execution thread to obtain the top-most execution context causing the execution chain, which then ended up in a potentially harmful operation. If the call originally was initiated by a trusted first-party functionality, the value is passed unaltered to the respective API or property. If the original call came from a third-party script, the value is automatically sanitized, so that no additional script content is added to the web document. In particular, this protection mode is useful if a codebase already exists that cannot be adjusted to use the unsafe API variants, e.g., legacy code that includes minified components.

3.2.2 Dangerous APIs

The purpose of ScriptProtect is to ensure that vulnerabilities in third-party code cannot be leveraged for an XSS attack. Hence, our security policies are set such that creation of additional script content cannot occur, whereas other types of content injection, e.g., including iframes or images hosted by other domains, are outside of our threat model. In essence, these considerations result in a policy which strictly *forbids third-party JavaScript code to conduct string-to-code conversions*, i.e., the introduction of additional executable JavaScript code that was derived or referenced from potentially untrustworthy data. For one, ScriptProtect prevents third-party code from creating additional `script` tags. Furthermore, a given third party is not allowed to introduce code via the inline event handlers of newly introduced HTML elements. The creation of an `iframe` with a `srcdoc` attribute or `javascript:` URI is prevented, as otherwise code execution on the origin of the first party can be achieved. Finally, ScriptProtect also strips third-party code from the ability to conduct direct string-to-code conversion via APIs like `setTimeout`. There exist a variety of different APIs and DOM properties which have the ability to introduce new code through one of the means above. These APIs and properties can further be divided into different classes of functionality, types of access and accept different kinds of inputs. In the remainder of this section, we briefly describe the characteristics of each of the classes of functionality including with table for all their representatives. Each of these needs to be considered by the ScriptProtect implementation to cover all possible cases through which vulnerabilities can accidentally be introduced by a third-party.

Raw DOM content at rendering time For one, JavaScript offers a set of APIs which add additional HTML code to the document directly during the initial rendering of the document. The code is seamlessly interpreted right after the script terminates and before subsequent HTML code is parsed/interpreted. The best known and most widely used API here is `document.write`. As Table 3.1 shows, the only alternative function of this class is the very similar `document.writeln`.

Table 3.1: APIs and properties that introduce new JavaScript code at rendering

Functionality	Type	Sink
document.write	Global API	HTML
document.writeln	Global API	HTML

Runtime creation of DOM content The next relevant class are DOM APIs and properties that allow the alteration of the HTML content at runtime, i.e., after the initial rendering process has terminated. Unlike `document.write` this functionality is not provided as a global API. Instead, it is achieved through a set of properties and APIs which are directly attached to DOM elements. This way, the relative location of the new content is provided implicitly. The most used element in this class is the `innerHTML` property, which on assignment inserts new HTML subtree-structures as a DOM child of the hosting HTML element. However, this class also includes all means to modify existing DOM elements, e.g. modifying the `src` attribute of a `script` tag as Table 3.2 summarizes.

Direct code conversion Apart from the addition of HTML markup to the document, JavaScript code may also be executed directly. In particular, a set of APIs and DOM properties allow the direct conversion of string data into JavaScript code. The most used representative of this class is the function `eval`, all other variants are listed in Table 3.3.

3.3 Protection Methodology

On a high level, ScriptProtect is a JavaScript library that instruments dangerous APIs with HTML sinks in such a way that all values passed to them are *sanitized*, before the API's DOM altering functionality is executed. In the context of this chapter, the process of sanitizing HTML input is regarded as an orthogonal problem and we assume that a safe implementation exists. At the time of writing in 2019, the browser did not yet provide a native way to sanitize JavaScript values for safe DOM inclusion, therefore we leveraged

Table 3.2: APIs and properties that can introduce new JavaScript at runtime

Functionality	**Type**	**Sink**
Element.innerHTML	Property	HTML
Element.outerHTML	Property	HTML
Element.setAttribute	Local API	all
Element.insertAdjacentHTML	Local API	HTML
HTMLScriptElement.src	Property	URI
HTMLScriptElement.text	Property	JS
HTMLScriptElement.textContent	Property	JS
HTMLScriptElement.innerText	Property	JS
HTMLIFrameElement.src	Property	URI
HTMLIFrameElement.srcdoc	Property	HTML
HTMLTag.on*EventName*	Property	JS
Range.createContextualFragment	Local API	HTML

Table 3.3: APIs and properties that can introduce new JavaScript code through direct code conversion. Asterisks denote non-standard APIs.

Functionality	**Type**	**Sink**
eval	Global API	JS
Function	Global API	JS
setTimeout	Global API	JS
setInterval	Global API	JS
setImmediate*	Global API	JS
execScript*	Global API	JS

the established DOMPurify [102, 52] library for this purpose. Combined with the fact that ScriptProtect is implemented as one single JavaScript file, this results in a lightweight approach that is easy to deploy and does not rely on custom modifications to the browser, which means ScriptProtect could be used immediately.

We achieve our measurement and protection by wrapping all functionality that could lead to code execution so that each call to an unsafe API is intercepted by our hook. Therefore, ScriptProtect is a JavaScript instrumentation that is implemented through monkey patching as introduced in Section 2.3.1. This also means that ScriptProtect needs to be the first script that is loaded into a page to ensure all following code automatically uses the secured APIs. One advantage is that this instrumentation is completely transparent to the rest of the application: pre-existing code which intends to call the original API keeps functioning without requiring any code changes. Moreover, we can use the same instrumentation to enforce our security policies in the modified dangerous APIs also to report all attempted violations during our compatibility measurements. If there are no violations during benign interaction, then the site is compatible with ScriptProtect. This means that by describing our instrumentation for the protection mechanism, we also implicitly describe the *scanning methodology* for our measurements in the next section, which solely relies on this instrumentation. Each of three types from Tables 3.1 to 3.3 requires a different instrumentation strategy, as we describe in the following.

3.3.1 Instrumentation of HTML Sinks

Global APIs The reference to global APIs such as `document.write` is readily available after document initialization and is very similar to the classic example of monkey patching JavaScript functions, as introduced in Section 2.3.1. We first preserve a link to the original implementation for future usage with sanitized values. Subsequently, we overwrite the global reference and replace it with a reference to a function which first executes the sanitizing step on all arguments, before calling the original functionality to achieve a transparent instrumentation.

Local APIs In the case of the instrumentation of element-local APIs that are directly attached to DOM node, no single global reference to an API exists as each individual DOM element exposes the API to the calling code. In this case, we have to leverage JavaScript's prototype-based object oriented features. All DOM nodes are decedents of JavaScript's `Element` object class. Thus, via altering `Element`'s prototype, we are able to change the behavior of *all* DOM nodes transparently. Fig. 3.1 shows this process. Specifically, we first get a reference to the original `insertAdjacentHTML` method in line 2. Subsequently, we overwrite the method in the prototype and only invoke the original variant after sanitizing the HTML markup passed as the second parameter to `insertAdjacentHTML`. For APIs that exist only for a subclass of DOM nodes, such as the `createContextualFragment` API of `range` elements, we instrument the respective, more specialized prototype.

```
(function() {
    let old = Element.prototype.insertAdjacentHTML;
    Element.prototype.insertAdjacentHTML = function() {
        if (arguments.length == 2) {
            arguments[1] = sanitize(arguments[1]);
        }
        return old.call(this, ...arguments);
    }
})();
```

Figure 3.1: Transparent instrumentation of local APIs

Properties DOM properties, on the other hand, cannot be instrumented directly. Instead, their setter property has to be replaced with the safe wrapper. As the DOM properties themselves are again attached to individual DOM nodes, we have to change the elements' prototype, similar to the element-local APIs. Figure 3.2 shows how we achieve this. `Object.defineProperty` allows us to overwrite the `set` property, i.e., the setter to be called when assigning a value to `innerHTML`. We then proceed to sanitize the input and invoke the original, stored setter.

```
(function () {
    var old = Element.prototype.innerHTML;
    Object.defineProperty(Element.prototype, "innerHTML", {
        set: function (val) {
            val = sanitize(val);
            old.call(this, val);
        }
    });
})();
```

Figure 3.2: Transparent instrumentation of properties

3.3.2 Instrumentation of JS and URI Sinks

JS sinks For APIs like `eval`, which directly result in code execution of the complete string, there exists no harmless subset of inputs. This is also true for the assignment of DOM properties that take JavaScript code as a parameter like `script.innerText`. Therefore, calls to these APIs with a JS sink do not involve a sanitization step, as shown in Figure 3.3. Instead, the call is completely blocked unless the introduction of new code was allowed through one of the two mechanisms described in the following sections.

```
(function () {
    let old = window.setTimeout;
    window.setTimeout = function() {
        //Allow only if there is no string-to-code conversion
        if (typeof arguments[0] != "string") {
            old.call(window, ...arguments);
        }
    };
})();
```

Figure 3.3: Instrumentation of `setTimeout`

URI sinks The same is also mostly true for the assignment of a URI, e.g. `script.src`, with the only exception that including further scripts from the host of the same third-party could be allowed as this domain is already trusted. However, allowing the inclusion of same-site scripts has some subtle drawbacks: For one, if the third-party is a CDN, then the attacker could abuse this to include older versions of libraries hosted on that CDN, allowing potential script gadget attacks [143]. Moreover, these might contain publicly disclosed vulnerabilities, which then could re-enable old attacks on patched and up-to-date sites. Furthermore, it is notoriously difficult to correctly identify the effective top-level domain. A script hosted on `*.amazonaws.com` should not trust other hosts on `amazonaws.com`, as they are all part of Amazon's public cloud infrastructure and anyone can obtain a subdomain for this domain to host malicious scripts. This could be solved by using the *Public Suffix List* (PSL) [80], which includes a list of both official ICANN suffixes, e.g. `co.jp`, and private suffixes like `cloudfront.net` or those for Amazon's AWS. With over 20,000 entries the list is rather large and weighs about 200KB, which is about 10 times the size of ScriptProtect itself and would have a negative impact on the loading time. For these reasons, we decided to block the assignment of URIs in general and treat these sinks in the same way as the JS sinks.

3.3.3 Unsafe API Variants

As previously discussed, ScriptProtect's measures create API variants that are *secure by default*. This means that if no script-reenabling steps are taken it is impossible to introduce additional JavaScript content into the web document. However, first-party code might require this capability occasionally. In this case, the majority of the application code uses the standard – now safe – APIs. Only in selected cases, in which the introduction of further script code is explicitly intended, a second, newly introduced variant of the API is called which allows the potentially unsafe action. The occurrence of such cases is most likely seldom and can be audited thoroughly, as the *insecurity is now explicit*. Implementation-wise, for global APIs we just attach the original, native function to a global object outside of our instrumentation closures under a new name like `document.unsafeWrite`. In case of local DOM properties, additional properties are added to the respective element's prototypes as shown in Fig. 3.4 at the example of `innerHTML`.

```
(function () {
  var oldSet = Object.getOwnPropertyDescriptor(Element.prototype, "innerHTML").set;
  Object.defineProperty(Element.prototype, "unsafeInnerHTML", {
    set: oldSet
  });
})();
```

Figure 3.4: Introduction of an unsafe innerHTML property variant

3.3.4 Dynamic Access Control

One important goal for the design of ScriptProtect was to also offer a protection mode that works *without any changes to existing applications*. Thus, instead of rewriting an application to use the unsafe API variants as discussed in Section 3.2.1, we offer an alternative protection mode based on dynamic access control. In this scenario, whether or not a call to a problematic API is allowed, depends on the party that induced that call in the first place. Calls that originate from the first party are then routed to the original, unaltered API while all calls induced by a third party will use the safe, instrumented APIs and properties.

By inspecting the current execution thread through the `Error` object, we can obtain the stack trace from within JavaScript code at runtime without requiring an external debugger. We then proceed to extract the URL of script at the top of this call stack, as shown in Figure 3.5. The script at the top of the stack trace represents the initiator of the actions that lead to the call in the first place. If the hostname of the script's URL matches the first party the call is allowed without modification. Otherwise, the function argument is subjected to value sanitization or blocked, depending on the sink.

```
function isAllowed() {
    //Extract all URLs from the stack trace
    var regex = /(https?:\/\/.+?):\d+:\d+/g;
    var urls = (new Error).stack.match(regex);

    if (urls && urls.length > 0) {
        //Use last entry and extract its hostname from URL
        var topCaller = getHost(urls[urls.length - 1]);
        return topCaller == location.hostname;
    }
    return true;
}
```

Figure 3.5: Code snippet showing how stack trace is parsed and the top caller extracted

3.4 Large-Scale Study

In this section, we want to answer three important questions about our approach:

- *Compatibility:* How many websites can use ScriptProtect without any changes?
- *Effectiveness:* How many of the discovered Client-Side XSSes would it prevent?
- *Performance:* How large is the impact to page load performance for website visitors?

3.4.1 Data Collection

To analyze the compatibility and effectiveness of ScriptProtect in the Alexa Top 5,000 we let our instrumented browsers crawl these web applications. However, we found that the Alexa list contains 103 `google.tld` domains and a total of 82 subdomains of `tmall.com`. To gravitate our analysis to a more diverse set of web applications we opted to skip those entries in the list for which we either already had a site included which has the same eTLD+1 or the same second-level domain. Additionally, we remove any entry for which we are unable to connect to the website according to the following pattern `http://ENTRY`. After this preparation step, we arrive at a new list of 5,000 sites to be crawled. Our crawlers follow each same-site link up to depth 2 with a maximum of 1,000 unique links per site. This allows us to analyze the web applications in more depth than the previous approaches [144, 156] and leaves us with around 3.5 Million pages on 4528 different sites.

On the remaining 472 sites, however, we were unable to visit more than one link successfully. Investigating these cases reveals that for 106 sites we were unable to connect to the main site via HTTP due to the connection being preemptively terminated (e.g., connection resets, unresolvable hostnames) or the site needing more than 30 seconds to load which triggers a timeout in our infrastructure in order to prevent infinite loading sites. Randomly sampling 5% of the other 366 reveals that on 9 sites our crawlers were blocked visiting the site and were instead served a static site indicating the block, with 1 location and 3 IP-based blocks. Further 6 sites only served static content such as CDN main sites and domain selling sites, and 3 sites would have required to circumvent interstitial JavaScript dialogs, e.g., GDPR interstitials. Of these remaining 4528 sites with more than one successfully visited page, only 65 do not include any third-party script on their Web presence. Thus, for our further analyses in the rest of this section, we focus only on these 4463 sites which could theoretically benefit from ScriptProtect.

In a separate study, we used taint tracking to search for Client-Side XSS flaws in the same set of websites. As this part of the evaluation was conduced by Marius Steffens and Ben Stock, we only briefly summarize the results here and refer the interested reader to our ScriptProtect publication [177] for more details. During this study, we found a Client-Side XSS on 351 sites, of which 129 sites are vulnerable due to scripts originating from third-party hosts. Moreover, on 37 of these, we also found a vulnerable flow originating from the first-party code, leaving us with 92 (26%) sites, which are *solely* vulnerable due to third-party code.

3.4.2 Website Compatibility

Ideally, ScriptProtect is used when creating new applications. Then, the unsafe API variants clearly indicate which code parts could lead to vulnerabilities, aiding both manual and automated security analysis. Still, existing applications can also profit from the protection today without requiring any code changes by using the trace-based inspection of calls. A site is compatible as long as all included third-party scripts do not add additional JavaScript on their own during normal operation, i.e., through external scripts, inline scripts, or event handlers. In this case, ScriptProtect would not block any action of the third party during a normal visit without an attempted attack — the site could add our protection without any changes and without breaking existing functionality.

By artificially injecting the protection via our instrumented browser and visiting these sites and all their subpages in the set of the initial 3.5 Million pages, we observe that on the vast majority of sites, a third party dynamically adds new code at least once. This is due to the fact that constructing and assigning script URLs at runtime is extremely popular, e.g., in advertisements, and used by third parties on about 94% of sites. On the other hand, only 70% of these sites include a third party, which uses APIs and properties with an HTML sink like `innerHTML` *to insert new JavaScript code*, triggering the sanitization step to block that new code. Furthermore, only 35% sites have a third-party which use methods with a direct JavaScript sink like `setTimeout`. Our results on the usage of the different sinks are summarized in Table 3.4.

Table 3.4: Number of sites for which a third-party used a sink in order to add new code. See Tables 3.1 to 3.3 for a mapping of APIs and properties to sinks.

Description	**# of sites**
Successfully crawled	4528 (—)
With third-party scripts	4463 (100%)
Third-party adds code via URI sinks	4180 (94%)
Third-party adds code via HTML sinks	3122 (70%)
Third-party adds code via JS sinks	1562 (35%)

Coming back to the 129 sites that are vulnerable due to flaws in third-party code, we investigated which sink was actually responsible for the vulnerability in the first place. We find that 122 of the 129 sites were exploitable due to the injection of HTML markup, 4 due to an injection into `eval`, 2 sites had an injection into `script.src`, and on another 2 sites the attacker can hijack the content of a `script.text` attribute. One site had both an injection into an HTML context and into eval, resulting in 130 sinks on 129 sites. This shows that the APIs and properties with a JS or URI sink are preventing compatibility with a significant amount of existing sites, while the real-world vulnerabilities are only rarely caused by them. Intuitively this makes sense, as the direct assignment of a `script.text` or call to `eval` makes it very obvious to the developer that new code is created. On the other hand, for example, a call to `innerHTML` to adjust the content of a `div` tag does not make its security implications completely obvious, again highlighting the need to make the insecurity explicit. Therefore, to achieve backward-compatibility with a larger number of existing applications, we activate our instrumentation only for the HTML sinks, which still mitigates most risks. With our approach of only instrumenting the HTML sinks, ScriptProtect can be used on 1341 of the 4463 sites with third-party code *without any code changes*.

3.4.3 Mitigation Effectiveness

To verify that this backward-compatible version of ScriptProtect indeed provides the targeted protection, we artificially added our `scriptprotect.js` to the top of the `head` of all pages in the set of the 129 sites with vulnerable third-parties. Subsequently, we checked whether the proof-of-concept exploits discovered by the taint engine were blocked by our protection mechanism. Due to the design choice to exclude them, the 8 sites with non-HTML sinks could not be protected. However, another 13 also continued to be vulnerable, as their third-party scripts were vulnerable to an *HTML injection*, but then proceeded to insert our payload without using one of the protected *HTML sinks*. Manual investigation of these 13 sites showed that this is due to the fact that, when using certain libraries, the line between the different sinks begins to blur, as also previously discussed in Section 2.2.2. For example, jQuery's `.html` function is a more convenient version of `innerHTML`, that internally uses `script.text` (or `eval` in older versions) to execute inline scripts, which is not possible using the standard `innerHTML` function. As jQuery is widely used we extended ScriptProtect to also wrap and protect all of jQuery's HTML sinks. Adjusting our protection to correctly function with all other popular libraries would have required manual analysis of each and thus was considered out of scope, but would be straightforward on a case-by-case basis. After adding the additional protection of jQuery another 6 sites were protected, leaving only 15 sites vulnerable despite ScriptProtect's presence.

Overall, this backward-compatible version of ScriptProtect prevents the exploitation of the discovered third-party vulnerabilities on 114 of the 129 sites. While a more complete protection certainly would be desirable, a trade-off between security and compatibility needs to be made. With our approach of only instrumenting the HTML sinks, ScriptProtect can be used on 30% of the sites with third-party code *without any code changes* while still preventing almost 90% of all third-party caused vulnerabilities.

To understand if other approaches could mitigate the risk of an exploitable flaw in a similar fashion, we also checked if the sites in question could deploy a strict Content Security Policy (CSP). In doing so, we found that *all* sites with a vulnerable third-party script also made use of inline scripts or event handlers. Therefore, these sites could not easily deploy a secure policy without modifications; at the very least, securing inline scripts would require to use nonces. For event handlers, though, there was a discussion about allowing these via a new CSP keyword called `unsafe-hashed-attributes`. While the feature eventually shipped under the name `unsafe-hashes`, at the time of writing there was no solution other than using the `unsafe-inline` keyword, which would entirely undermine CSP's protection capabilities. Hence, the only way to use CSP would be to rewrite large parts of the application [270]. We see this as a further evidence that ScriptProtect indeed fills the much needed gap of an easy-to-deploy mechanism that helps to prevent Client-Side XSS attacks.

3.4.4 Runtime Performance

As ScriptProtect is an always-on mechanism that adds additional access-control checks to important APIs we expect it to have some performance impact during runtime. To evaluate this we randomly sampled 50 different pages from a set of compatible sites with

Table 3.5: Performance measurements of ScriptProtect, showing the time in milliseconds until the load event fired.

	Avg. Median	Std. deviation	Slowdown
Baseline	1280	1278	-
ScriptProtect	1360	1377	1.06

the condition that these specific pages included at least one third-party script. We used a 2015 Macbook Pro with a 2,2 GHz Intel i7 processor, 16 GB RAM, running Mac OS X 10.14.1 and Google Chrome 70.0.3538 for these experiments.

After an initial visit to populate the cache each page was visited 20 times: 10 times completely unmodified to establish a baseline and 10 times with ScriptProtect enabled. After all these visits we took the median for each of the two configurations to be more resistant against outliers caused by the network or remote server. ScriptProtect was locally injected by us, but we argue that after the first page load it would be loaded from cache anyway. The final overall results were obtained by averaging over the medians of all 50 pages and are shown in Table 3.5.

The minified version of ScriptProtect used in the evaluation consists of 19 KB and increases the load time by about 6%. However, our proof-of-concept relies on DOMPurify to sanitize inputs and with 14 KB most of the size is from this library, while ScriptProtect itself only weighs about 5 KB. Consequently, most of the increase in load time is caused by the parsing and initialization of the script itself. Fortunately, in the current standardization of trusted types for the DOM [96] the introduction of a browser-native sanitizer is on the roadmap [97]. Thus, as soon as this functionality is available directly in the browser, ScriptProtect also loses DOMPurify's network traffic and parsing time. In addition, it is to be expected that a native sanitizer vastly outperforms any JavaScript solution, further adding performance improvements.

3.5 Related Work

In the following, we discuss how ScriptProtect relates to previous papers in the area of defenses against different types of Cross-Site Scripting and previous work on the secure inclusion of untrusted code.

3.5.1 XSS Defenses and Mitigation

Given that XSS has been around for almost 20 years, a number of researchers have proposed defenses and mitigations against these attacks. Early research focused on deploying such tools on the server, such as Vogt et al. [261] or Bisht and Venkatakrishnan [22]. Later, Ter Louw and Venkatakrishnan [249] proposed *BluePrint*, a tool enabling Web sites to provide a specification of the expected DOM structure; this way, any anomalous script content could be easily identified and removed. This approach, however, requires changes to the browser itself, which our approach does not.

In 2010, Bates et al. [17] analyzed the security of existing browser-based XSS detection. In doing so, they found a number of flaws in Internet Explorer's filter, which would even allow for XSS in error-free Web sites. Based on their insights, they proposed a new concept for an XSS filter, dubbed *XSSAuditor*. This filter is nowadays deployed in all Webkit-based browsers, such as Google's Chrome. However, as Stock et al. [242] showed in their work, design choices in the XSSAuditor make it susceptible to bypasses: in 2014, 73% of the sites with vulnerabilities carried at least one flaw for which the Auditor could be bypassed. Pelizzi and Sekar [197] proposed an improvement variant with more aggressive filtering, naturally accompanied by an increased chance of false positives. Hence, the improved changes were not applied to the original Auditor.

In the area of defenses against Cross-Site Scripting, Stock et al. [242] proposed to use taint tracking to stop code injections. They extended their taint engine to forward taint into the JavaScript parser and enforcing policies to ensure that user-provided data could only be interpreted as literals and not lead to code execution. While their approach protects against all types of Client-Side Cross-Site Scripting, it requires immense changes to the codebase of the browser and causes an overhead between 7% and 17%. Consequently, this has not been implemented into browsers as of now.

3.5.2 Securing Third-party Code

Over the years there have been many attempts to allow the inclusion of third-party code without compromising the security of the including site itself. In 2007, Jim et al. [121] proposed *BEEP*, a browser-enforced embedded policy that controls which scripts are allowed to run. Similar to today's CSP, it allows whitelisting of specific scripts by including their SHA1 hash in the policy. For a more fine-grained approach, Meyerovich and Livshits [158] designed *Conscript*, which allows for specific security policies that are added to each script tag. Possible policies could, for example, forbid the use of dynamic scripts or calls to specific functions like `postMessage`. In a similar fashion, Van Acker et al. [256] created *WebJail* in 2011. While certainly powerful, these approaches require drastic modifications of the browser to enforce the policy and, without adoption by popular vendors, have not found widespread use. Finally, Snyder et al. [228] built a browsing extension that works in a similar fashion to our hooking approach. In particular, this enables a user to selectively disable DOM features which are not needed by a site. In contrast, however, in our work we do not require a user with an extension or a study what a given site requires to function correctly. More importantly, though, our approach allows the continued usage of all DOM APIs, yet securing access to them.

In a different kind of approach, Miller et al. [161] created a subset of JavaScript called *Caja*, an object-capability language which isolates objects from the outside world. However, this means that all untrusted code must be written in *Caja*. Following the same concept, Agten et al. [5] proposed *JSand* in 2012, a system that isolates third-party scripts from each other and the DOM through the use of an object-capability model. In the same year, Ingram and Walfish [115] presented *Treehouse*, a JavaScript sandbox based on web workers. While these and similar approaches can be implemented without modifications to the browser, they require changes to the untrusted code. Hence, adoption of this type of defense is inhibited by the lack of support from the third-parties like advertisement networks.

Finally, there are defensive mechanisms which are implemented as transparent wrappers, so that neither the browser nor existing code needs to be modified. In 2009, Phung et al. [201] published their work on self-protecting JavaScript, in which they intercept security relevant events by monitoring the methods and fields of built-in objects. One year later, Ter Louw et al. [248] proposed *AdJail*, an isolation framework specifically designed for advertisements. The isolation is achieved by running the code in a so-called shadow page and by providing a controlled interface for interaction with the real page. Yet, all these papers assume a malicious third-party in their attacker model and hence require very strict isolation of the third-party code. This, in turn, tends to break many use cases like analytics and enrichment of pages in general. In contrast, our concept is much more lightweight and simpler to integrate. In particular, there is no need to tamperproof our hooked functions, as we block all untrusted code *before* it is executed.

Still, there also has been some work on securing benign-but-buggy third parties, confirming the relevance of our threat model: In 2015, Weissbacher et al. [271] presented their system *ZigZag*, which transparently instruments JavaScript code to perform anomaly detection during runtime. After an initial learning phase, the generated models are used to harden the client-side code.

4 Discovering Abuses of JavaScript Capabilities

In an ideal world, all security analyses would be fully automated and never require human intervention. Clearly this unrealistic and in practice the manual inspection of a potentially malicious website is sometimes still necessary. When discussing the different instrumentation approaches in Section 2.3, we already introduced the *DevTools* that act like an IDE and can be attached to any tab to inspect and analyze its content. Unfortunately, by running JavaScript code on a malicious website, attackers have various ways to mess with the results of such an inspection. Even more worryingly, some techniques can detect the mere *presence* of the DevTools, therefore enabling attackers to exhibit different behavior when under analysis, thus invalidating all results. In this chapter, we investigate these abuses of JavaScript capabilities that prevent or at least slow down, any attempts at manually inspecting and debugging of a website and call them *anti-debugging techniques*. Therefore, in contrast to the previous chapter where we used the scanner to detect vulnerabilities and evaluated a potential defense, these anti-debugging techniques will serve as an use case for our web security scanning pipeline to detect ongoing attacks in the wild.

In Section 4.1, we first describe the manual analysis of websites and thus the general attack scenario in more detail. Moreover, we outline our threat model and explain what we consider in and out of scope for this study. Then, we provide a description and systematization of 9 anti-debugging techniques in Section 4.2, which we divide into 6 *basic* and 3 *sophisticated* techniques. As we consider these techniques as an interesting use case for our modern security scanner, we then focus on discovering such abuses automatically, i.e., through an instrumented browser as part of our scanning pipeline. Thereby, we also run challenge of *volatile content* from Section 2.2, as some techniques only become apparent when visiting the website multiple times. We describe how we address this challenge, as well as our detection methodology for each of the 9 anti-debugging techniques in Section 4.3. Then, we report the results of two studies on anti-debugging techniques on 1 million websites in the wild in Section 4.4 and conclude with a review of related work in Section 4.5.

4.1 Use Case: Anti-Debugging Techniques

We use our browsers to visit new websites almost every day, some of which might not be trustworthy at all. Nevertheless, we visit them and execute their JavaScript code on our computers, while relying on the browser to keep us safe. Yet browsers are incredibly complex applications, e.g., in 2020 the Chromium browser had over 25M lines of code [23]. Unsurprisingly, some of these lines have bugs that can have severe security implications [e.g., 56, 55, 57, 54]. Therefore, detecting and analyzing JavaScript malware is a crucial task to maintain the security of the Web platform.

Heavy obfuscation and the ability to generate new code during runtime makes a fully static analysis of malicious JavaScript largely infeasible. Therefore, effective detection often relies on a dynamic analysis or a combination of both [e.g., 51, 213, 53, 126]. This then led to a shift towards *evasive* malware which abuses implementation differences between a real browser and dynamic analysis systems, leading in turn to new approaches to deal with such evasive techniques [127]. Yet one, so far, overlooked scenario is the manual analysis of websites using a normal browser, since we can only combat evasive malware deceiving our automated tools if we can manually inspect and learn from it. Unfortunately, this scenario opens up new paths for inventive attackers to interfere with the analysis by creating *anti-debugging techniques targeting humans using real browsers.*

4.1.1 Manual Analysis of JavaScript Code

While previously developers and malware analysts might have relied on browser extensions such as FireBug [188] to inspect a website, nowadays all browsers ship with powerful, integrated Developer Tools [88], or *DevTools* for short. At the time of writing the DevTools of Chromium shipped with 24 different tabs, each focusing on a different feature. In the following, we will briefly introduce the four most useful of these features.

The *elements* tab shows the DOM tree of the currently displayed page. It automatically updates all elements if JavaScript code manipulates them and all elements can also be changed by the user and directly affect the rendered page. The *sources* tab not only allows the inspection of the whole client-side code but also includes a full debugger. With it, the user can set breakpoints anywhere, step through the code, inspect the call stack and variable scopes, and even change the value of variables on the fly. The *console* tab acts like an interactive shell, which allows you to execute arbitrary JavaScript code in the top-level scope of the currently loaded page. If execution is currently suspended at a breakpoint, all code executed in the console will run in the scope of the breakpoint's location instead. The *network* tab, like the name suggests, allows full inspection of all network traffic including the headers and timing data. On top of that, the DevTools offer many advanced features like measuring site performance with a stack-based profiler, creating a heap snapshot to investigate memory leaks, and the ability to measure and inspect code coverage.

Using any other analysis tool that is not part of a browser, e.g., static analysis or executing a single script in isolation is usually *not* an option if one wants to obtain reliable results, due to multiple reasons: First of all, JavaScript code written for the Web expects many objects that are not part of the language specification, like `document` or `location`. Moreover, scripts often load additional code on the fly, e.g., one particular script might generate code for an iframe with a URL as the source and add that to the DOM. The browser then requests the content for that iframe over the network, which might contain additional script code which then again loads additional code via an `XMLHttpRequest`. Previous research has shown that such patterns of deep *causality trees* in script inclusions are a common occurrence today [138, 137]. Only a real browser is able to correctly handle the inherent complexity of modern Web applications and thus only a real browser can be used to accurately inspect and analyze JavaScript code on the Web.

4.1.2 Threat Model and Scope

Throughout this chapter, we consider the following scenario: A user, also referred to as the analyst, manually visits a given website in a real browser to analyze and interact with the website's code. In particular, the user intends to browse the source code of that website, set breakpoints and step through the code, and inspect variables and functions. On the other hand, the website does not want to be analyzed and contains evasive measures to detect and hinder or, at least, slow down and deter any attempts at inspection.

We consider the browser's integrated DevTools the tool of choice for the user to achieve their analysis goals. As previously outlined, the DevTools are not only full of useful features, but with their integration into the browser also the only way to correctly execute the JavaScript code in the first place. Moreover, using them also avoids the problem of evasive malware potentially detecting the inspection by noticing it does not run in a real browser.

In scope In general, the underlying problem in this scenario is that the analyst can not fully trust the capabilities used during a live inspection, e.g., any logged output during execution, as the website might have manipulated the logging functionality on-the-fly. Furthermore, if the website is able to detect the presence of the inspection, it could also alter or completely suppress any malicious activity to appear benign during analysis. In this chapter, we investigate all these techniques that affect the *dynamic analysis* of a website, like altering built-in functions or detecting the presence of a debugger. We refer to such techniques as *anti-debugging techniques* from now on.

Out of scope Since we only focus on techniques that are affecting the code at runtime, all *static code transformation* techniques, in particular obfuscation, are out of scope. While these can certainly be a powerful tool to greatly slow down manual analysis, especially when combined with some of the anti-debugging techniques introduced in the following, these static techniques have already been extensively studied in the past [e.g., 32, 275, 274, 74]. Similarly, all techniques that do not affect a real browser but rather aim to break sandboxes or other analysis systems, e.g., by intentionally using new features or syntax not yet supported by these systems, are out of scope as well.

4.2 Systematization of Techniques

In this section, we will introduce 6 *basic anti-debugging techniques (BADTs)* and, after that, 3 *sophisticated anti-debugging techniques (SADTs).* These techniques each have one of the following three different goals: Either to outright impede the analysis, or to subtly alter its results, or to just detect its presence. During its introduction, we will give each technique a short name, e.g., *ModBuilt*, by which it will be referenced throughout the remainder of the paper and will also provide a link to a mention of this technique on the Web. Additionally, we provide a testbed available at `https://js-antidebug.github.io/` with one or two exemplary implementations for each technique so that the interested reader can experiment with each technique while reading this chapter. Moreover, we will also briefly describe possible countermeasures for each BADT to give a better impression of how effective they are. We conclude with a systematization of all techniques that summarizes this section.

4.2.1 Basic Anti-Debugging

The first three BADTs all just try to impede attempts at debugging the website. They are generally not very effective but still might cause an unsuspecting user to give up in frustration.

Preventing Shortcuts (ShortCut) Before any meaningful work can begin, the analyst first needs access to the full client-side code of the website and thus the following BADT simply tries to prevent anyone from accessing that source code. The quickest way to open the DevTools is by using a keyboard shortcut. Depending on the browser and platform there are multiple slightly different combinations to consider, e.g., for Chrome on Windows `F12`, `Ctrl+Shift+I`, and `Ctrl+Shift+J` all work. As JavaScript has the ability to intercept all keyboard and mouse events as long as the website has the focus, these actions can be prevented by listening for the respective events and then canceling them, as shown in Figure 4.1 [232]. This obviously can not prevent someone from opening the DevTools by using the browser's menu bar.

Besides the advanced DevTools, common browsers also have a simple way to just show the plain HTTP response of the main document. This can usually be accessed by right-clicking and selecting "View page source" from the context menu, or directly with the `Ctrl+U` shortcut. Again, both these actions can be prevented by listening for these events and then canceling them. There are many ways to easily bypass this, e.g., by prefixing the URL with the `view-source:` protocol or opening the sources panel of the DevTools.

Triggering breakpoints (TrigBreak) The `debugger` statement is a keyword in JavaScript that has the same effect as manually setting a breakpoint [68]. As long as no debugger is attached, i.e., the DevTools are closed, the statement has no effect at all. This behavior

```
window.addEventListener("keydown", function(event){
    if (event.key == "F12") {
        event.preventDefault(); return false;
}});
```

Figure 4.1: Disabling the F12 shortcut

makes the statement a perfect tool to only interfere with debugging attempts. The technique can be as simple as just calling the debugger in a fast loop over and over again. As a simple measure to counter this technique, the DevTools of popular browsers have the option to "Never stop here", effectively disabling only the `debugger` statements while still allowing breakpoints in general. However, many variations exist which make it harder to reliably block it, e.g., constantly creating new anonymous functions on the fly instead of always hitting the breakpoint at the same location [224]. On the other hand, this can still be countered by specific code snippets that remove all `debugger` statements on the fly, like the *Anti Anti-debugger* script [94] for the Greasemonkey browser extension [9].

Clearing the Console (ConClear) While the sources panel for the DevTools offers the ability to inspect and change variables in the scopes of the current breakpoint, the console can be very useful in this regard as well. For example, it allows one to easily compare two objects or to run a simple statement at the current location of the suspended execution. However, it is possible to make the console unusable by constantly calling the `console.clear` function [233]. If done fast enough, this makes it near impossible to inspect the output and thus the value of variables during runtime without setting breakpoints with the debugger. However, this technique can be circumvented by enabling "Preserve log" in the DevTools options or by disabling the clear function by redefining it to an empty function.

Instead of only blatantly trying to impede the analysis, the following technique can also subtly alter what an analyst observes during debugging attempts.

Modifying Built-ins (ModBuilt) As JavaScript allows monkey patching, all built-in functions can be arbitrarily redefined. For instance, a popular music streaming service for a while had modified the `alert` function, which many bug bounty hunters use to test for XSS, to secretly leak all client-side attempts to trigger an XSS attack to their back-end, as shown in Figure 4.2.

As this example demonstrates, the possibilities to redefine built-in functions and objects to make them behave differently are basically endless. Furthermore, there are many legitimate use cases, like polyfills that provide a shim for an API not supported by older browsers. Since we are only interested in functions that a human analyst is likely to use in the DevTools console, we focus our search on modifications to the `console`, `String` and

```
// Wrapping funcs in a naive attempt to catch externally found XSS vulns
(function(fn) {
  window.alert = function() {
    var args = Array.prototype.slice.call(arguments);
    _doLog('alert', args);
    return fn.apply(window, args);
  };
}(window.alert));
```

Figure 4.2: This code including the comment was found on spotify.com in 2018 [269]. The `_doLog` function reports the current URL along with a full stack trace to their backend any time the `alert` function is called.

`JSON` objects, and their respective functions. Figure 4.3 shows a somewhat contrived example of how malicious code could hide itself [76]. Note that this technique can also be used to impede the analysis instead, e.g., by redefining all functions like `log` and `info` to an empty function [224, 235]. A possible countermeasure is to save a reference to every native function one intends to use before executing any of the malicious code, a tactic popular in JavaScript rewriting and sandboxing literature [e.g., 201, 177].

```
let original = console.log;
console.log = function(arg) {
  if (arg == "shellcode") { arg = "benign code"; }
  original(arg); }
```

Figure 4.3: Redefining the log function to hide malicious code

Finally, the most subtle of all techniques only try to detect the presence of the analysis. In contrast to the previous technique which directly altered the behavior of built-in functions an analyst would use, these techniques instead aim to alter the control flow of their own code. This way, attackers could suppress executing malicious code for any user that opens the DevTools or had them previously open on the same domain.

Inner vs. OuterWidth (WidthDiff) By default, opening the DevTools either splits the browser window horizontally or vertically. In JavaScript, it is possible to obtain both the size of the whole browser window including all toolbars (outer size) and the size of the content area without any toolbars (inner size). Thus by constantly monitoring the `outerWidth` and `innerWidth` properties of the window object, we can check if the DevTools are currently open on the right-hand side. The same works if the DevTools are attached to the bottom, by comparing the height instead, as shown in Figure 4.4. This is the method used by the popular *devtools-detect* package [226] that, at the time of writing, already had over 1000 stars on Github and is thus probably often used in the wild. This is also the technique used by the credit card skimming case [220] from the introduction.

```
setInterval(() => {
    if (outerWidth - innerWidth > threshold ||
        outerHeight - innerHeight > threshold) {
        //DevTools are open!
    }
}, 500);
```

Figure 4.4: Monitoring the window size to detect the DevTools

However, this technique does not work if the DevTools are undocked, i.e., open in a separate, detached window. Additionally, this technique will report a false positive if any other kind of sidebar is opened in the browser.

Log Custom Getter (LogGet) Exactly because of the just described drawbacks of the WidthDiff technique, some developers are interested in more reliable alternatives. A StackOverflow question titled "Find out whether Chrome console is open" [231] back from 2011 so far received 130 upvotes and 14 answers. While many of the suggested approaches have stopped working over the years, some answers are still regularly updated and present working alternatives.

In particular, for at least the last three years, some working variations of what we call the LogGet technique existed. The technique works by creating an object with a special getter that monitors its `id` property and then calling `console.log` on it. If the DevTools are open, its internals cause it to access the id property of every logged object, but if they are closed, the property is not accessed. Therefore, this getter was a reliable way to determine if the DevTools are open. While the original approach stopped working sometime in 2019, someone created a variation of it that uses `requestAnimationFrame` to log the element with the custom getter which still works as of time of writing. As an alternative, it is also possible to overwrite the `toString` function of an arbitrary function and then log that function, as shown in Figure 4.5. Since the DevTools internally also use toString to create the printed output, we know that the DevTools are opened whenever this toString function is called.

```
var logme = function(){};
logme.toString = function() {
    //DevTools are open!
}
console.log('\%c', logme);
```

Figure 4.5: Approach from 2018 to detect the DevTools

As long as one of these variations continues to work, this method is a very reliable way to detect if the DevTools are open, as it also works if they are detached or already open when the website is loaded. There is no real countermeasure except to remove all logging functions of the console object, an invasive step which by itself also might get detected.

4.2.2 Sophisticated Anti-Debugging

In contrast to the BADTs seen so far, the following *sophisticated anti-debugging techniques (SADTs)* are much more elusive. The following three SADTs are *timing-based*, i.e., they rely on the fact that certain operations become slower as long as the DevTools are open. On a high level, these techniques get the current time, e.g., via `Date.now` or `performance.now`, perform some action and then check how much time has passed. If that time is above a specified threshold or changes significantly at one point, then the DevTools were likely opened. These techniques thus use the time between operations as a side-channel about the state of the DevTools. Firefox, for example, lowers the resolution of timers due to privacy concerns and to mitigate side-channel attacks like Spectre [154]. Yet a precision in the range of milliseconds is still more than enough for these techniques to work.

Monitor existing Breakpoint (MonBreak) As the debugger statement only halts the execution if a debugger is attached, we can simply compare the time directly before and after that statement. If it took longer than, e.g., 100ms then we can be sure that the DevTools are open [76]. Figure 4.6 shows how this technique can be implemented in a few lines of JavaScript code. The main difference to TrigBreak is that the goal here is not to disrupt the user but rather to infer the state of the DevTools. So, in this case, triggering the breakpoint only once is already enough to know somebody is analyzing the website and there is no need to trigger additional breakpoints afterward.

```
function measure() {
    const start = performance.now();
    debugger;
    const time = performance.now() - start;
    if (time > 100) { /*DevTools are open!*/ }
}
setInterval(measure, 1000);
```

Figure 4.6: Detecting the DevTools by checking for an attached debugger

Wait for new Breakpoint (NewBreak) A more stealthy variation of the MonBreak technique does not trigger breakpoints by itself, but rather detects when the analyst is adding a new breakpoint anywhere. As soon as this new breakpoint is hit, we can again observe this through timing information. If we call a function repeatedly in the same interval and suddenly it took way longer to execute again, there is a good chance that a breakpoint was hit. While this approach is more stealthy, it obviously has no effect as long as someone uses the DevTools without setting a breakpoint at all. Also, note that `setInterval` and similar functions are throttled if the user switches to another tab. Therefore, an additional check with hasFocus is needed to confirm that this page is currently in the foreground, as shown in Figure 4.7.

```
function measure() {
    const diff = performance.now() - timeSinceLast;
    if (document.hasFocus() && diff > threshold) {
        //DevTools are open!
    }
    timeSinceLast = performance.now();
}
setInterval(measure, 300);
```

Figure 4.7: Detecting the DevTools by checking the time between multiple executions

Console spamming (ConSpam) While the debugger statement is a useful tool to implement anti-debugging measures, it still has the drawback that halting at breakpoints can easily be disabled in the DevTools. The following technique instead abuses the fact that certain functions of the browser-provided window object run slower while the DevTools are open. Historically, this worked by creating many text elements with long content and quickly adding and removing them to the DOM over and over again [231]. This caused a noticeable slowdown, as the elements tab of the DevTools tries to highlight all changes to the DOM in real-time. However, this approach no longer works in both Firefox and Chrome. What still works, at the time of writing, is to write lots of output to the console and check how long this took [98]. As the browser needs to do more work if the console is actually visible, this is a useful side-channel about the state of the DevTools. Conveniently, this technique also works regardless of which tab in DevTools currently has the focus.

Figure 4.8 shows a possible implementation of this ConSpam technique. An alternative is to first measure the time a few rounds in the beginning and then always compare to that baseline. This has the advantage that a visitor with slow hardware does not trigger a false positive, as there is no fixed threshold. However, this approach then assumes the DevTools are going to be opened after the page has loaded and not right from the start.

```
function measure() {
    const start = performance.now();
    for (let i = 0; i < 100; i++) {
        console.log(i);
        console.clear();
    }
    const time = performance.now() - start;
    if (time > threshold) { /*DevTools open!*/ }
}
setInterval(measure, 1000);
```

Figure 4.8: Detecting the DevTools by repeatedly calling functions of the console

4.2.3 Summary

To put all 9 techniques into context, we examine them based on four properties: Effectiveness, stealth, versatility, and resilience. An *effective* technique has a high likelihood of activation and thus causing an impact on the analyst. As such, LogGet is an effective technique while ShortCut might never really affect anyone. A *stealthy* technique wants to remain unnoticed, i.e.,WidthDiff is a stealthy technique (although the measures it takes upon detection of the DevTools might be not so stealthy) while TrigBreak is the very opposite of stealthy. A *versatile* technique can be used to achieve many different outcomes, as opposed to something very specific. Therefore, ModBuilt is a versatile technique as it can redefine a built-in function to anything else and LogGet can react in many different ways if it detects the DevTools. For the same reason, all SADTs are versatile since they only detect the presence of the analysis and do not prevent the use of certain features. A *resilient* technique is not easily circumvented, even if the user is aware of its existence. For example, LogGet is a resilient technique because there is no good countermeasure, while DevCut was easily bypassed by using the menu bar. While MonBreak stops working if breakpoints are disabled, the other two SADTs are rather resilient since they are hard to disarm unless one finds their exact location in the code. Table 4.1 shows the full results of our systematization for each technique. As all four properties are desirable from the perspective of an attacker, the techniques LogGet and ConSpam offer the most potential.

Table 4.1: Systematization of anti-debugging techniques. A filled circle means the property fully applies, a half-filled circle means it applies with limitations.

Technique	**Goal**	**Effective**	**Stealthy**	**Versatile**	**Resilient**
ShortCut	Impede	○	○	○	○
TrigBreak	Impede	●	○	○	◐
ConClear	Impede	◐	○	○	○
ModBuilt	Alter/Impede	◐	◐	●	◐
WidthDiff	Detect	◐	●	●	○
LogGet	Detect	●	◐	●	●
MonBreak	Detect	●	○	●	○
NewBreak	Detect	◐	●	●	◐
ConSpam	Detect	●	◐	●	●

4.3 Scanning Methodology

The previously mentioned *devtools-detect* package [226] and also the question on Stack-Overflow [231] already indicated a certain interest in anti-debugging techniques, in particular in detecting whether the DevTools are open. However, so far, there has not been a comprehensive study on the prevalence of these techniques in the wild. In this section, we will therefore present a fully automatic methodology to detect each of the BADTs from the previous section. This methodology will then be used in the next section to conduct a measurement of anti-debugging techniques in the wild.

In the following, we will briefly outline how we can detect the presence of each BADT during a single, short visit to the website. After that, we will explain our methodology to detect SADTs, which is more generic and thus also more complicated and requires multiple visits to the page. As in previous chapters, we are again relying on a real Chromium browser instrumented via the CDP. This way we can, for example, inject JavaScript into each context before the execution of any other code occurs or programmatically controlling the behavior of the debugger, which are both very useful to detect anti-debugging techniques.

4.3.1 Methodology for BADT Detection

To detect BADTs, we use the fact that all these basic techniques have an obvious "signature" that is easy to detect, e.g., logging an object with special properties. While the detection methodology presented in this section is specifically tailored to each technique and only able to detect exactly them, this methodology is simple, effective, and scales very well.

ShortCut To detect intercepted key presses, we first collect all event listeners via the `getEventListeners` function. For each collected `keydown` or `contextmenu` listener, we create an artificial keyboard or mouse event to imitate the shortcut or right click. We pass this event to the listener and then check if the `defaultPrevented` property of the event was set, i.e., the respective normal behavior was blocked by this listener.

TrigBreak By registering the `Debugger.paused` event of the DevTools protocol, we can observe the location of each triggered breakpoint. We log this data and immediately resume execution, to not reveal the presence of the debugger itself.

ConClear To check for attempts at constantly clearing the console, we first register a callback to the `Runtime.consoleAPICalled` event of the DevTools protocol. This API notifies us of all invocations of functions of the `console` object and thus allows us to observe how often `console.clear` is called.

ModBuilt We inject JavaScript code into each website which is guaranteed to execute before any of the website's code. Our injected code then creates a wrapper around each object and all of its properties we want to observe. This wrapper will notify our back-end if someone overwrites them or one of their properties. We ignore code that only adds new properties that do not overwrite existing functionality, e.g., a polyfill that adds a new function like `String.replaceAll` to browsers that do not yet support this feature.

WidthDiff We use a similar wrapper as described in ModBuilt, only this time we monitor for read accesses instead of writes to the property `innerWidth` and its siblings. Since we expect that tracking and fingerprinting scripts, in particular, might be interested in some of these values to determine the screen resolution of all visitors, we only flag scripts that access all four properties.

LogGet Similarly to the ConClear technique, we observe all interactions via the console APIs. As the technique requires one to log some specifically crafted objects that are unlikely to be logged during normal operations of a website, we can look out for those. Thus, if we observe a format string logged together with a function that has a custom toString function like in Figure 4.5, we flag the page. The same applies if we observe the logging of an object that has an `id` property which is a function instead of a value.

Triggering breakpoints or clearing the console once or twice is rather harmless, they only become a problem if they happen constantly. Therefore, for all these 6 BADTs we not only detect *if* they happen but also *how often* per script. One disabled shortcut could be a coincidence, but disabling all five within the same piece of code is most likely a deliberate attempt at preventing access to the source. For this, we only count occurrences in the main frame of the loaded page, since (usually rotating) advertisements should not influence the numbers. Moreover, many techniques lose their effectiveness in iframes, e.g., ShortCut would only prevent the shortcut while the iframe is focused. We aggregate all numbers by site, i.e., if a given technique is present on multiple subpages of the same site, we only count it once. The same applies if one site has multiple different scripts that trigger the same technique. In all cases, we only use the most significant occurrence of each technique within a site for further analysis, e.g., the script that cleared the console most often.

4.3.2 Methodology for SADT Detection

The main challenge in detecting SADTs is that they are a lot more flexible and thus not as easy to detect as the BADTs. In particular, we can not identify them by just monitoring a few specific function calls and property accesses. While all SADTs rely on timing information, they do not necessarily need access to the `Date` or `performance` objects, as they could also get a clock from a remote source, e.g., via WebSockets. Therefore, we need a more general approach to reliably detect sophisticated techniques in the wild. In the following, we will describe how we address this challenge and then report on our findings.

While these SADTs can differ in how they are implemented, they still have something in common: They try to figure out whether they are currently analyzed or not and then behave accordingly. Therefore, code execution must diverge from the default, benign case as soon as the analysis is detected. If we somehow could monitor the executed code twice, once with the DevTools open and once with them closed, and then compare those two executions, we would be able to isolate the SADT. Thus, our methodology is based on two concepts: *deterministic website replay* and *code convergence*.

Deterministic website replay To obtain meaningful results when visiting the same website multiple times, we first need a way to reliably load it exactly the same way. In particular, this means we do not want the server-side logic to have any influence on the response and we also do not want dynamic content like different ads on every page load. Therefore, we must load the website only once from the remote server and cache all content on a local proxy. Afterward, we ensure that our browser can not connect to the outside world and loads the page only from our proxy to avoid any interaction with the remote server. However, we also must disable all ways to obtain randomness on the client-side. Otherwise, if a URL parameter contains a random id, the proxy will not have seen this request before and be unable to answer as expected. Thus we replace `Math.random` with a PRNG implementation with a fixed seed and use a fixed timestamp as a starting point for all clock information in `Date` and `performance`.

In theory, without any external logic or randomness, the page should behave entirely deterministic every time we load it, which is exactly what we need for our analysis. Unfortunately, the replays are not entirely perfect. Since in the browser and also the underlying operating system many actions are executed in parallel, the exact order of events is not always deterministic. For example, consider a website with multiple iframes which all send a postmessage to the main frame upon completion. The main frame could execute different code depending on which frame loaded first. So even if our replay system otherwise works perfectly, we can not prevent that small performance differences in this multi-process system sometimes cause one iframe to load faster than another, leading to different behavior in the main frame in the end. Getting rid of these performance fluctuations is unrealistic, as it would require immense changes to both browser architecture and the underlying operating system's scheduler. Therefore, we instead rely on the concept of *code convergence* to deal with this problem.

Code convergence The idea here is that the more often we replay the same website, the lower the likelihood becomes that we will discover any new execution paths caused by small timing differences. Or to describe it more briefly: The executed code converges over time. We thus replay each page multiple times and always measure the code coverage, i.e., we track which statements in a script are executed and which are not, across all scripts on the page. By merging all seen code from the previous replays, we can check if the current replay introduced any new statements. In the same way, we can also build the intersection of all previously executed code and check if some parts were not executed, which always had been executed before. If now, for multiple replays, no new code is added nor always executed code missing, we likely have executed until convergence.

By combining these two concepts, we can now replay any website in the same environment until convergence. We can then inject analysis artifacts into the page, like attaching a debugger or adding a breakpoint. As long as we do not make any changes to the website's code, it should behave like during the previous replays. This means we should not see any completely new code, nor should code be missing that previously was always executed. If, however, we reliably observe different code execution only when our analysis artifacts are present, then these differences are most likely caused by an anti-debugging technique.

Implementation We implemented our approach as a tool that can detect SADTs in a fully automated fashion. As in our first study, we control the browser from Node.js by using the *Chrome DevTools Protocol (CDP)*. This protocol exposes all features of the DevTools for programmatic access and gives us low-level information and callbacks for many useful events. In particular, the CDP gives us fine-grained code coverage data with the `Profiler.takePreciseCoverage` command. Moreover, the protocol lets us control the debugger, so we can programmatically enable breakpoints and set them at specific locations, which we need to detect MonBreak and NewBreak. Since the CDP does not include a way to open the DevTools on demand, we instead cause an artificial slowdown of the console to detect the ConSpam technique. We implemented this by wrapping all functions of the `console` object to first execute a busy loop for a short time, which approximates the slowdown normally caused by an open DevTools window.

For the replaying part, we use a modified version of *Web Page Replay (WPR)* [90], a tool written in Go that is developed and used by Google to benchmark their browser. The tool is designed to record the loading of a website and creates an archive file with all requests and responses, including the headers. This archive file can then be used to create a deterministic replay of the previously recorded page. WPR also tries to make the replays as deterministic as possible, by injecting a script that wraps common sources of client-side randomness like `Math.random` and `Date` to always use the same seed values. Additionally, we improved the accuracy of the replays by extending WPR to always answer with the same delay as the real server during the recording. By combining our Node.js browser instrumentation and this modified Go proxy, we can now automatically detect anti-debugging techniques in the wild.

4.4 Large-Scale Study

In this section, we first describe our results from a large-scale study on 1 million websites in the wild. As the detection methodology for SADTs does not scale enough to use on such a large number of pages, this first study is limited to the detection of BADTs. Then, we also present our findings from a second, targeted study of SADTs on the websites with the most severe BADTs discovered during the first study.

4.4.1 Large-Scale Study of BADTs

Experiment setup For our study on BADTs, we visited the 1 million most popular websites according to the Tranco list [139] generated on 21 Dec 2020. We started 80 parallel crawlers using Chromium 87.0.4280 on 22 Dec 2020 and finished the crawl three days later.

On each page, our crawler waits up to 30 seconds for the load event to trigger, otherwise we flag the site as failed and move on. After the load event, we wait up to 3 more seconds for pending network requests to resolve to better handle pages which dynamically load additional content. Finally, we then stay for an additional 5 seconds on each loaded page, so that techniques that take repeated actions like TrigBreak or WidthDiff have enough time to trigger multiple times.

Of all the sites of the initial 1 million, about 15% could not be visited at all, despite having used the most recent Tranco list. Of these, about 8% were due to network errors, in particular, the DNS lookup often failed to resolve. In another 4%, the server returned an HTTP error code and the remaining 3% failed to load before our 30 seconds timeout hit. In total, we successfully visited around 2.8M pages on about 846k sites, where site refers to an entry in the Tranco list which then consists of one or more pages. We did not only visit the front page because research on the *cryptojacking* phenomenon has shown that a common evasive technique is to not run any malicious code on the front page to avoid detection during brief inspections. In line with previous research [134, 178], we therefore additionally selected three random links to an internal subpage and visited these as well.

Prevalence First of all, we are interested in the general prevalence of BADTs in the wild. As can be seen in Table 4.2, we can find indicators of behavior resembling the six BADTs on over 200k sites. The overwhelming majority of these are caused by ModBuilt and WidthDiff, which, judging from the high numbers, seem to be common behavior also in benign code. Moreover, we can see that visiting subpages did indeed significantly increase the prevalence by about 17% compared to only crawling the front pages. Interestingly, indicators of the more desirable techniques (using the properties from our systematization in Table 4.1) are also more often hidden in subpages. Specifically, TrigBreak is a clear outlier here and breakpoints occurred a lot more often only on subpages.

Table 4.2: Number of sites with indicators for each technique and the increase from also visiting subpages.

Technique	# Websites	% Total	# Subpages only
ShortCut	4525	0.53	818 (+22%)
TrigBreak	1128	0.13	502 (+80%)
ConClear	3061	0.36	981 (+47%)
ModBuilt	101587	12.00	15345 (+18%)
WidthDiff	114154	13.49	18615 (+19%)
LogGet	3044	0.36	756 (+33%)
Total	206676	24.42	30494 (+17%)

These results in Table 4.2 should only be seen as indicators for behavior resembling those of the six BADTs. Next, we analyze how *confident* we are for each occurrence that it is used in an intentional and malicious manner. As previously stated, there is a huge difference between clearing the console once and clearing it 50 times within a few seconds. On the other hand, it makes little difference anymore if it is cleared 20, 50, or even 1000 times which are all highly unusual and hard to cause by accident. In between those two extremes, there is a window of values that are suspicious but not definitely malicious, e.g., clearing it 5 times. As Figure 4.9 shows, for many techniques about 50% of all detections were caused by just a single occurrence. Looking at ConClear, we can see that of all sites that cleared the console at least once, only about 4% cleared it between 6 and 10 times and only 1% cleared it more than 10 times.

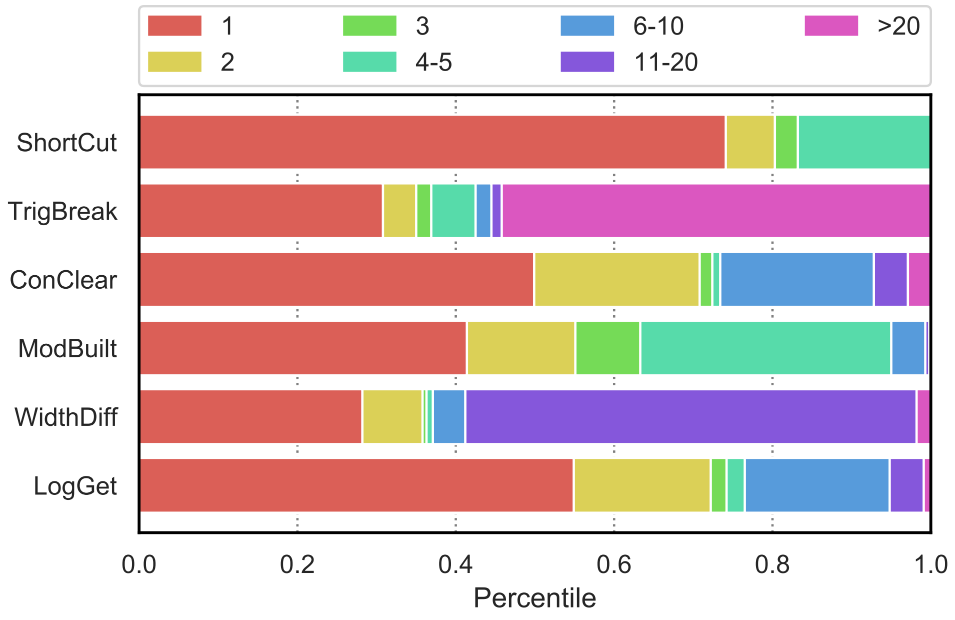

Figure 4.9: Occurrences within each BADTs grouped into 7 bins, e.g., all sites on which a technique triggered 11-20 times share the same bin. The bins are only used for data visualization and not for further analysis.

To compare indicators of different techniques, we first need a normalized value that incorporates these insights from Figure 4.9. Therefore, we calculate the *confidence score* by taking the squared value of the percentile within that technique. For example, if we visit a site and see one script that clears the console twice, we would assign a confidence score of $0.6^2 = 0.36$ to this script. On the other hand, if the same script would trigger 30 times, we would assign a confidence score of $0.95^2 = 0.9025$ to it. The rationale behind this formula is that the percentile encodes how often the number of occurrences was observed compared to observations of the same technique on other websites. Squaring this value then puts more weight on the unusually high occurrences, e.g., when the console is cleared dozens of times, resulting in a higher confidence that this usage is intentional and resembles anti-debugging efforts.

Yet, we still have to consider that clearing the console is by itself an uncommon occurrence, with only about 0.53% of all sites behaving this way. A ConClear event with low confidence can still be more significant than e.g.,ModBuilt with a higher confidence score. Thus, we next calculate a *severity score*, which combines the confidence score with the inverse frequency of the techniques, i.e., the more common a technique the less it increases this score. For this, we use the *Inverse Document Frequency* (IDF) from the domain of information retrieval and adapt it to count techniques instead of word terms. Thus, the weights for each technique are calculated as follows: ln(number of sites with any technique / number of sites with given technique). This means that the presence of ConClear has a weight of 4.21 while ModBuilt only has 0.71. We then multiply the confidence score with these weights and build the sum over all techniques on the site to obtain the final severity score. Overall, this score considers that 1) some techniques are rarer than others, 2) some sites use these techniques more aggressively than others, and 3) combining different techniques on the same site is more effective.

Results Based on our severity score, we can now analyze the most significant cases of anti-debugging in more detail. In this and the following sections, we focus on the 2,000 sites with the highest severity score, which represents approximately the top 1% of all sites with any indicators. These sites all had a severity score of 3 or higher, as shown in Figure 4.10. Moreover, the same figure shows that more than two-thirds of these sites had multiple BADTs on the same site, with a few sites as many as 5 simultaneously. On average, the severity score on these 2,000 sites was 4.63 and the average amount of techniques on the same site was 2.28, as the raw numbers in Figure 4.11 show.

First, we wanted to see if there is a correlation between the popularity of a website and the prevalence of BADTs. We investigated this separately for each technique, to account for their high variance in the total number of occurrences. As shown in Figure 4.12, BADTs were slightly more prevalent in the higher ranking and thus more popular websites, with the notable exception of ShortCut.

Next, we analyzed the *code provenance* of the scripts we found to be responsible for executing the BADT by distinguishing between first- and third-party scripts, for which we used the *eTLD+1*. The *eTLD* is the effective top-level domain, e.g., for foo.example.co.jp the eTLD is .co.jp and thus the eTLD+1 is example.co.jp which corresponds to the "registrable" domain. However, it should be noted that the following analysis based on the eTLD+1 is only a rough estimation. For example, a third-party library could also be hosted on first-party servers or first-party code on another domain like a CDN which then would appear to be third-party code. In general, it is rather complex to correctly determine if multiple domains belong to the same owner, as previous research has shown [e.g., 137, 150, 254, 238].

Now as Table 4.3 shows, we get a very different picture depending on the technique: ShortCut was mainly caused by first-party code, while ModBuilt was more balanced. On the other hand, WidthDiff showed the exact opposite and was with an overwhelming majority present in third-party code. But even if a technique was triggered by third-party

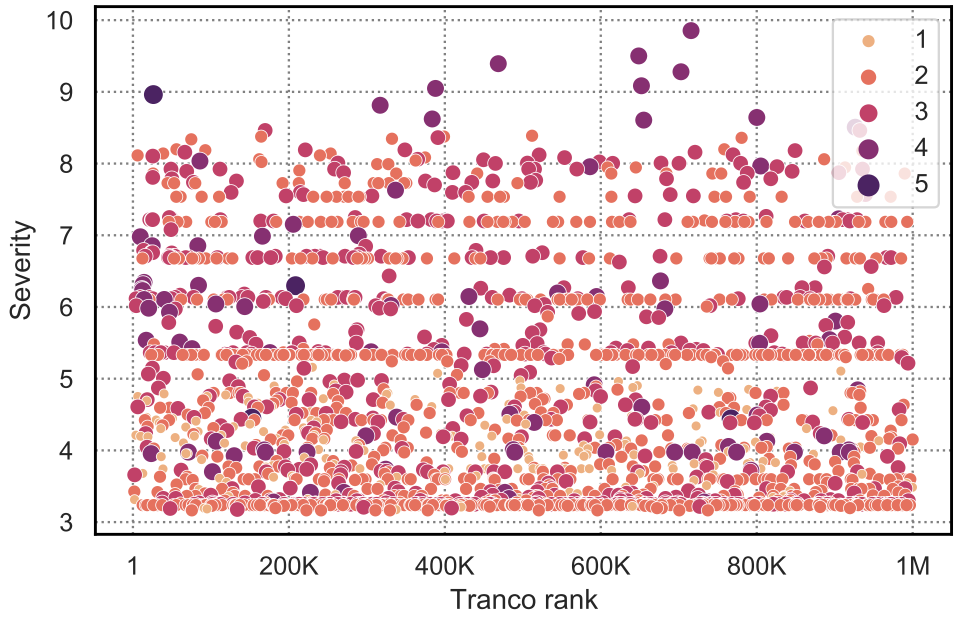

Figure 4.10: Scatter plot showing the distribution of the severity scores over the Tranco ranks. Size and color both indicate the number of simultaneous techniques on the website.

code, it still can very well be the first party's intent to interfere with an analysis by including their code. For example, the most prevalent script for causing both ShortCut and TrigBreak in third-party code is a plugin for the popular e-commerce platform Shopify called *Vault AntiTheft Protection App* [69], which promises to protect the website from competitors that might want to steal one's content.

Now we next want to know if the number of third-party inclusions is caused by relatively few popular scripts or not. In Figure 4.13 we can see that, e.g., for WidthDiff the most popular script is already responsible for about 51% of all cases in third-party code and the top 5 together cover already 77%. This means that only a very small number of scripts is responsible for the high prevalence of this technique, while for other BADTs this behavior is

Figure 4.11: Severity scores on the left and sites with multiple techniques on the right.

Severity	# Sites
3-4	1095 (54.75%)
5-6	563 (28.15%)
7-8	330 (16.50%)
9-10	12 (0.60%)

(a) Severity scores

Combo	# Sites
1	201 (10.05%)
2	1142 (57.10%)
3	565 (28.25%)
4	88 (4.40%)
5	4 (0.20%)

(b) Combinations of BADTs

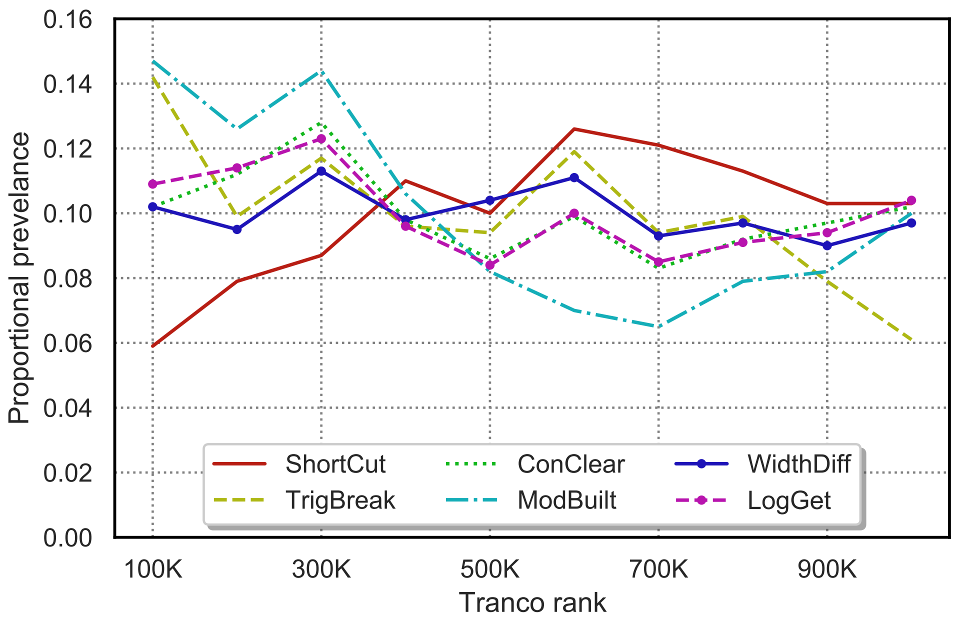

Figure 4.12: Normalized correlation between website rank and prevalence of each technique in 100k buckets. The most popular sites are on the left.

less pronounced. Moreover, LogGet and ConClear almost perfectly overlap each other, as the most popular implementations also try to hide the suspicious logged elements by clearing the console immediately afterward each time.

To further investigate this, we performed a manual analysis of the 10 most prevalent third-party scripts for each of the 6 BADTs. We found that many of these scripts are related to advertisements, bot detection, content protection, and cryptojacking. Moreover, many of them were not just minified but completely obfuscated. In total, 35 of the 60 most prevalent scripts and in particular 9 of the 10 most common scripts causing LogGet were obfuscated, indicating that these scripts would rather not be analyzed and might even be

Table 4.3: BADT occurrence by first- and third-party code.

Technique	**# First-party**	**# Third-party**
ShortCut	283 (73%)	103 (27%)
TrigBreak	282 (81%)	68 (19%)
ConClear	221 (17%)	1084 (83%)
ModBuilt	145 (43%)	195 (57%)
WidthDiff	19 (3%)	707 (97%)
LogGet	197 (16%)	1059 (84%)
Total	1147 (26%)	3216 (74%)

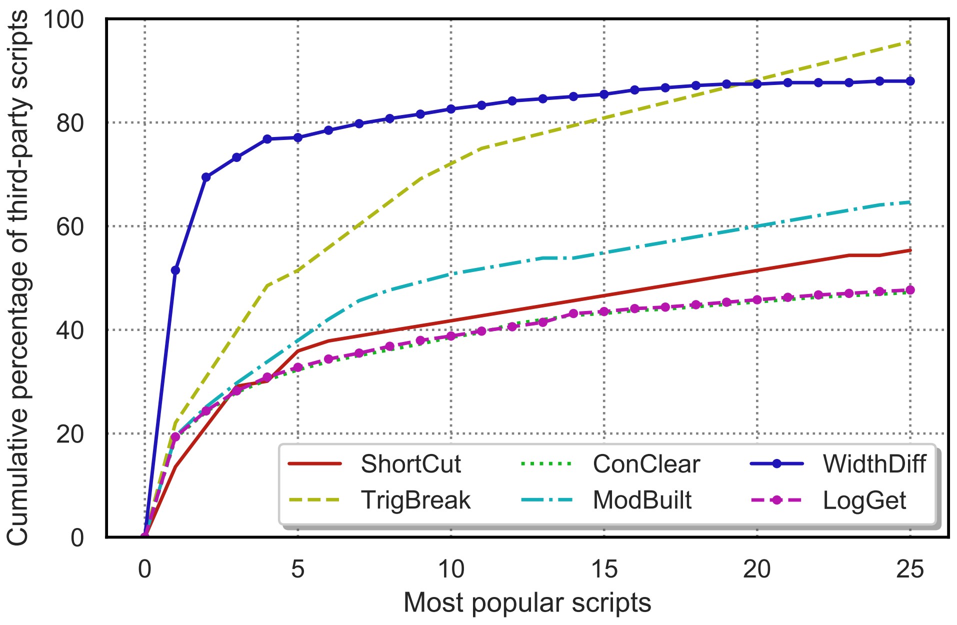

Figure 4.13: The 25 most common scripts for each technique and their cumulative share of sites

related to malicious activities.

4.4.2 Targeted Study of SADTs

Experiment Setup To get accurate results for our SADT detection, it is important to replay each page multiple times to ensure we have reached code convergence. Therefore, we record each website once and replay it until we get 10 consecutive measurements without any changes in coverage. If after 50 replays this still did not happen, we discard the website as being incompatible with our replaying infrastructure. After convergence, we test each technique 5 times. We only count a technique as present if it caused differences in at least 3 of the replays, to ensure their effect on the code coverage is reproducible.

Replaying this many times is a costly process, especially since we need to restart the browser with a new profile between each replay. Otherwise, stored state in cookies, local storage, and other places could lead to different execution branches. Therefore, in this second study, we only target the 2000 websites with the highest severity score according to our previous study on BADTs in Section 4.4.1. In the following, we will investigate whether this score is also a good indicator for the presence of sophisticated techniques.

Prevalence While we started this study directly after the first had finished, nevertheless 33 out of the 2000 selected sites were no longer reachable. Another 6 sites did not converge even after 50 replays. On 229 out of the remaining 1961 sites, we could find behavior similar to one or more of the three SADTs. Thus, about 12% of these sites executed different code when under analysis.

Table 4.4: Sites with SADTs in first- and third-party code.

Technique	# All	# First-party	# Third-party
MonBreak	138	124	24
NewBreak	85	38	54
ConSpam	8	5	3
Total	229	165	81

As Table 4.4 shows, the MonBreak technique was the most common of the three and present on around 14% of the investigated sites. On the other hand, ConSpam was rather uncommon with less than 1% prevalence. The technique MonBreak was mostly seen in first-party code, while NewBreak was a bit more often seen in third-party code. However, any difference in third-party code execution might also cause differences in first-party code and vice versa. Thus, there is some overlap between first- and third-party code detections. When comparing these results to another sample of 100 randomly selected sites, we only found 1 site with a SADT, in this case NewBreak. We see this low false positive rate as evidence that our approach to detect sophisticated techniques is reliable. Furthermore, we can see that BADTs are indeed a good indicator for the presence of further sophisticated techniques.

4.5 Related Work

In this section, we will first present works on anti-debugging in native malware, followed by publications on the topic of malicious JavaScript in general and conclude with the most closely related papers about evasive malware on the Web.

4.5.1 Anti-Debugging in General

Anti-debugging techniques are a well-known concept from the area of native x86 malware. Back in 2006, Vasudevan and Yerraballi [259] proposed the first analysis system that focused on mitigations for anti-debugging techniques. Their system called Cobra can, in particular, deal with self-modifying and self-checking code and thus counters many anti-analysis tricks. In 2010, Balzarotti et al. [13] proposed a technique to detect if a malware sample behaves differently in an emulated environment when compared to a reference host. Their main challenge was to achieve a deterministic execution of the malware in both environments so that a robust comparison of behavior becomes possible. Therefore, they first record all interaction of the malware with the operating system to exactly replay the results of the system calls in the second run. One year later Lindorfer et al. [148] extended on this idea with their system called Disarm, by not only comparing the behavior between the emulation and a real system, but instead comparing behavior between four different emulation systems.

Kirat et al. [131] improved on these previous works by creating an analysis platform called *BareCloud* which runs the malware in a transparent bare-metal environment without in-guest monitoring. However, the cat and mouse game continued by finding new techniques to detect and evade even these bare-metal analysis systems. In 2017, Miramirkhani et al. [164] presented their work on "wear and tear" artifacts, i.e., detecting the analysis system because typical artifacts of human interaction with the system in the past are missing.

To summarize, we can see that the deterministic execution of malware in multiple environments and then comparing differences in execution is a well-established approach to analyze malware binaries. However, we are, to the best of our knowledge, the first to apply this concept for JavaScript code running in browsers and to provide insights into how wide-spread these techniques are in the wild.

4.5.2 Malicious JavaScript

Over the years, there have been many publications on malicious JavaScript in general without any particular focus on evasive measures or anti-debugging. Multiple works focused on drive-by attacks, e.g., *JSAND* by Cova et al. [51] uses anomaly detection combined with an emulated execution to generate detection signatures, while *Cujo* by Rieck et al. [213] use static and dynamic code features to learn malicious patterns and detect them on-the-fly via a web proxy. Similarly, *Zozzle* by Curtsinger et al. [53], uses mostly static features from the AST together with a Bayes classifier to detect malicious code. Targeting drive-by exploit kits, Stock et al. [243] presented their work on *Kizzle*. Their approach is based on the fact that while the obfuscated code of such attacks changes frequently, the underlying unpacked code evolves much more slowly, which aids the detection process. As a more general defense that is not based on a detector, Maisuradze et al. [149] proposed *Dachshund*, which removes all attacker-controlled constants from JavaScript code, rendering JIT-ROP attacks infeasible. Other works focused on malicious browser extensions [126], discovering evil websites [116], and creating fast pre-filters to aid the large-scale detection of malware [32, 73].

4.5.3 Evasive Malware on the Web

A few publications also specifically focused on evasive JavaScript malware, which actively tries to avoid being detected. In 2011, Kapravelos et al. [125] showed how they can detect the presence of a high-interaction honeyclient and subsequently evade detection. One year later, Kolbitsch et al. [133] created Rozzle, an approach to trigger environment-specific malware via JavaScript multi-execution. This way, they can observe malicious code paths without actually satisfying checks for browser or plugin versions. Improving on this, Kim et al. [130] presented their work on forced execution to reveal malicious behavior, with a focus on preventing crashes. To detect evasive JavaScript malware samples that evolve over time, Kapravelos et al. [127] designed Revolver, which utilizes similarities in samples compared to older versions of the same malware. Their rationale is that malware authors react to detections by anti-virus software and iteratively mutate their code to regain their stealthiness. In their work called *Tick Tock*, Ho et al. [108] investigated the feasibility of browser-based red pills, which can detect if the browser is running in a virtual machine from JavaScript code by using timing side-channels.

However, while these previous publications worked on the phenomenon of evasive Web malware, they all assume the malware is analyzed as part of an *automated system* and tries to detect differences in this analysis environment. On the other hand, our threat model instead considers *anti-debugging* measures to hinder or avoid detection by a *human analyst using a real browser*.

5 Studying Abuses Beyond JavaScript

In Section 2.2, we looked at the trend of *emerging technologies* and that new features are constantly added to the browser. Notably, as a result of the introduction of *WebAssembly* in 2017, JavaScript is not the only programming language anymore that the browser supports. While WebAssembly was designed with security in mind from the start, malicious actors nevertheless quickly abused this novel technology. This mainly happened as part of the *cryptojacking* phenomenon, which abuses the efficient computation offered by WebAssembly in a way that generates profits in the form of cryptocurrency for a malicious website operator. However, we could also discover attempts at abusing WebAssembly to obfuscate malicious code, as the new technology was mostly not yet supported by tools such as malware detectors and analysis frameworks. In contrast to the previous chapter which focused on abuses of JavaScript capabilities, this chapter will thus focus on malicious techniques enabled by the introduction of WebAssembly to the browser, which would have either been impossible or at least much less efficient when implemented in JavaScript.

In Section 5.1, we first briefly describe the concept of *cryptocurrenies*, which are relevant to understand for the cryptojacking attacks. We then introduce our threat model for WebAssembly abuses in general, as well as which types of mining we consider to be malicious and thus in scope. In Section 5.2, we then provide more technical background specifically about the *memory-bound* cryptocurrencies involved during web-based mining, the structure of WebAssembly modules, and how JavaScript and WebAssembly can interact with each other. Moreover, we also take a closer look at one popular mining implementation, to show which relevant browser features are involved during the mining. Then, we outline our detection methodology based on browser instrumentation in Section 5.3. In particular, we show how we can collected all WebAssembly modules while crawling the Web and describe our mining detection approach based on three phases. In Section 5.4, we present our results on the prevalence of cryptojacking attacks in the wild and also shown that existing countermeasures were insufficient. Based on a manual analysis of the collected modules, we also show for what other purposes the early adopters used WebAssembly for and discuss the discovered modules that used it for obfuscation. Finally, in Section 5.5 we present related work on the topic of cryptojacking and parasitic computing.

5.1 Use Case: Malicious WebAssembly and Cryptojacking

For a long time, JavaScript has been the only option to create interactive applications in the browser and especially the development of CPU intensive applications, such as games, has been held back by the subpar performance offered by JavaScript. As a remedy, several attempts to bring the performance benefits of native code to the Web have thus been made: Adobe has heavily promoted the *Flash platform*, Microsoft proposed *ActiveX*, and comparatively recently, Google introduced its *Native Client*. However, all these are tied to a specific platform and/or browser and could not gain acceptance on a large scale. While Adobe Flash marks an exception here, it suffered from a number of critical vulnerabilities over the years [195, 272], resulting in dwindling acceptance. By now all these technologies have been deprecated [160, 39, 4]. As already introduced in Section 2.2, WebAssembly instead has been published as a standardized and platform-independent alternative. Only a few months after its initial publication, it has been implemented in all four major browser engines [153] and gained traction ever since, as the low-level bytecode language allows for significantly faster transmission, parsing, and execution in comparison to JavaScript [47].

5.1.1 Cryptojacking and Web-based Mining

In a completely independent development from aforementioned browser features, cryptocurrencies such as Bitcoin and Ether have gained popularity in the last years as they provide an alternative to centrally controlled fiat money and a profitable playground for financial speculation. A basic building block of these currencies is the process of *mining*, in which a group of users solves computational puzzles to validate transactions and generate new coins of the currency [180]. Although the stability and long-term perspectives of cryptocurrencies are not fully understood, they have attracted large user communities that mine and trade coins in different markets with considerable volume. For example, Bitcoin reached an all-time high of 66,900 USD per coin in October 2021 [48], resulting in a market value comparable to major companies. Unfortunately, this development has also attracted miscreants who have discovered cryptocurrencies as a new means for generating profit. In a strategy denoted as *cryptojacking*, users are tricked into unnoticeably running a miner on their computers, so that malicious actors can utilize the available resources for generating revenue in the form of cryptocurrencies.

Web-based mining is a variant of cryptojacking that cleverly combines the previous two developments for malicious purposes. It works by injecting mining code into a website, such that the browser of the victim mines during the website's visit. This way, the efficiency enabled by WebAssembly is abused to covertly perform intensive computations in a novel form of parasitic computing, without requiring to infect the visitor with native malware. First variants of these attacks have emerged with the availability of the Coin-Hive miner in September 2017 [3, 135]. Although originally developed for benign purposes, CoinHive has been maliciously injected into several websites [e.g., 83, 95]. In August 2018, a vulnerability in MikroTik routers has been used to inject a cryptojacking script into traffic passing through more than 200,000 of these routers [49]. Apart from this specific mining technique, however, it remains unclear what WebAssembly is widely used for on the Web and whether other malware based on WebAssembly exists. In this chapter, we therefore not only provide a study on cryptojacking in the wild, but also conduct the first comprehensive and systematic investigation of the larger WebAssembly ecosystem.

5.1.2 Threat Model and Scope

WebAssembly can execute at near-native speed, but still runs in a memory-safe, sandboxed environment in the browser and is also subject to security mechanisms like the same-origin policy [263]. In this chapter, we investigate new attacks enabled by the introduction of WebAssembly that previously would have not been possible or at least much less effective as a mere JavaScript implementation. For this, we focus on abuses of WebAssembly that do not violate its designed security boundaries. In particular, trying to break out of the sandbox and the browser to attack the victim's machine directly are considered out of scope here.

On one hand, our threat model thus obviously includes cryptojacking in the form of web-based mining. Such activity certainly also has legit use-cases and may pose an alternative to online advertisements as scheme of monetization. Therefore, we define cryptojacking as the practice of *automatically* starting a web-based miner upon visiting a web page. For this, we neither consider the disclosure of the mining process to the user nor the presence of an opt-out mechanism relevant. We view a consent after the fact as an inadmissible mode of operation, similarly to how the GDPR now requires a "clear affirmative action" for tracking cookies in the EU [50]. Miners that only run after explicit consent by the user, such as Authedmine and JSEcoin, are not considered part of the problem and are thus not examined in our study. To conclude that a website employs cryptojacking, we further do not differentiate between scripts added by the website's owner and scripts injected by a third party by means of hacking the server or hijacking included scripts. On the other hand, this threat model also includes novel obfuscation techniques that try to circumvent malware detectors trained on JavaScript files [e.g. 213, 53, 73], which are mostly unable to detect malicious code hidden in WebAssembly modules [216].

5.2 Web-based Mining

The mining of classic cryptocurrencies, such as Bitcoin and Ether, requires specific hardware to be profitable. Therefore, these currencies are not suitable for web-based mining, as visitors of a website will not run their browsers on such custom hardware. In this section, we discuss how so-called *memory-bound* cryptocurrencies instead enable profitable mining even with of-the-shelf hardware. Moreover, we describe the structure of WebAssembly modules in more detail and how JavaScript code can interact with WebAssembly and vice-versa. Finally, we conclude this section with a closer look at one popular implementation of a web-based miner.

5.2.1 Memory-bound Cryptocurrencies

Hardware devices designed for demanding computations, such as GPUs and ASICs, provide a better mining performance than common CPUs. As a consequence, profitable mining of classic cryptocurrencies has become largely infeasible with regular desktop and mobile computer systems. This situation has not been anticipated in the original design of the first cryptocurrencies and violates the "one-CPU-one-vote" principle underlying Bitcoin mining [180]. As a remedy, alternative cryptocurrencies have been developed in the community that make use of memory-bound functions for constructing computational puzzles. One prominent example is the cryptographic mixing protocol *CryptoNote* [219] and the corresponding proof-of-work function *CryptoNight* [222]. CrypoNight is a hash function that determines the hash value for an input object by extensively reading and writing elements from a 2 Megabyte memory region. This intensive memory access bounds the run-time of the function and moves the overall mining performance from the computing resources to the available memory access performance. As memory access is comparably fast on common CPUs due to multi-level caching, CryptoNight and other memory-bound proof-of-work functions provide the basis for alternative cryptocurrencies that can be efficiently mined on regular desktop systems and hence are a prerequisite for realizing web-based miners.

The idea of memory-bound proof-of-work functions along with other improvements over the original Bitcoin protocol has spawned a series of novel cryptocurrencies, each forking the concept of CryptoNote. The most prominent example is Monero [250] with a market capitalization of 4.2 billion USD [48] as of 2021. Due to the concept of anonymous transactions it also provide more privacy than Bitcoin and may conceal the identity of senders and receivers [165, 136]. Profitable mining on desktop systems render this currency an ideal target for web-based mining. Furthermore, the increased privacy of transactions provides a basis for conducting cryptojacking over manipulated web sites.

5.2.2 WebAssembly Modules under the Hood

As already mentioned, WebAssembly is the other piece to the puzzle that enables profitable computations for web-based miners. In general, WebAssembly code is structured in modules, which are self-contained files that may be distributed, instantiated, and executed individually. A WebAssembly module consists of individual sections of 11 different types, such as `code`, `data`, and `import`. Subsequently, we briefly describe the four most relevant section types.

Code Section This usually is the largest section as it encapsulates all function bodies of the WebAssembly module. One example of a function in this code section can be found in Table 5.1. This example comprises only basic functionality, like control flow statements, addition, and multiplication. Of course, there also exist a number of instructions with more complex mechanics, such as `popcnt`, which counts the number of bits set in a number, or `sqrt`, which calculates the square root of a floating-point value [263]. Note that these instructions do not involve any registers, but operate on the stack only, which is based in the design of the underlying virtual machine.

Table 5.1: A simple C function on the left and the corresponding WebAssembly bytecode along with its textual representation, the *Wat* format, on the right-hand side [263].

C program code	Binary	Text representation
	20 00	`get_local 0`
	42 00	`i64.const 0`
	51	`i64.eq`
int factorial (int n) {	04 7e	`if i64`
if (n == 0)	42 01	`  i64.const 1`
return 1	05	`else`
else	20 00	`  get_local 1`
return n * factorial (n-1)	20 00	`  get_local 1`
}	42 01	`  i64.const 1`
	7d	`  i64.sub`
	10 00	`  call 0`
	7e	`  i64.mul`
	0b	`end`

Data Section The optional data section is used to initialize memory, similar to the `.data` section of x86 executables. Amongst others, this section contains all strings used throughout the module. Figure 5.1 shows the definition of such a data segment in Wat format. In this example, four bytes are saved at the memory offset 8, which results in the number 42 if read as an unsigned integer.

```
(data_segment
  0                           // memory index
  (init_expr (i32.const 8))   // byte offset
  (data 0x2a 0x0 0x0 0x0)     // the data itself
)
```

Figure 5.1: Initializing memory with the number 42.

Import and Export These sections define the imports and exports of a WebAssembly module. The import section consists of a sequence of imports, which are made available to the module upon instantiation, that is, any listed function can then be used via the `call` opcode. For importing a function the module name, function name, and its type signature needs to be specified. It is then up to the host environment, for instance the browser, to resolve these dependencies and check that the function has the requested signature [263]. The export section, in turn, defines which parts of the module (functions or memory) are made available to the environment and other modules. Everything declared in this section can also be accessed via JavaScript as well.

5.2.3 WebAssembly and its JavaScript API

WebAssembly is intended to complement JavaScript, rather than to replace it. Consequently, it comes with a comprehensive JavaScript API, that allows sharing functionality between both worlds and allows instantiating WebAssembly modules with only a few lines of JavaScript code. To speed up the initialization of a new module, the Wasm binary format is designed such that a module can be partially compiled while the download of the module itself is still in progress. To this end, the API provides the asynchronous `instantiateStreaming` function, to complement the older synchronous `instantiate` function. Of course, WebAssembly modules may also be instantiated from data, embedded in JavaScript code—for instance, as raw bytes in an `Uint8Array`.

```
const obj = {
    imports: {
        imported_func: function(arg) { console.log(arg); }
    }
};
const wasm = await WebAssembly.instantiateStreaming(
    fetch('example.wasm'), obj
);
let result = wasm.instance.exports.factorial(13);
```

Figure 5.2: Instantiating a WebAssembly module and calling an exported function.

One short example on how to instantiate a Wasm module and interact with it in only a few lines of code can be seen in Figure 5.2. In this example, the first parameter of the `instantiateStreaming` call uses the `fetch` function to load a module over the network. The second parameter is an object specifying the imports for the module and, in this example, exposes the `console.log` function to the WebAssembly environment. This is necessary to grant the Wasm code access to functions from the JavaScript domain. The same restrictions, for instance, also apply to access and modifications to the DOM. In last line, the exported function `factorial` is invoked to pass execution to the WebAssembly module. The corresponding Wasm code is the called, executed, and the value returned to the JavaScript environment. Alternatively, the `factorial` function could be changed to make use of the imported `console.log` functionality to directly print the value from within the Wasm module.

5.2.4 The CoinHive Miner

To get a better understanding on how web-based miners combine all these presented technologies, we take a closer look at the CoinHive miner [135], which was the most popular mining service at the time of writing, as we will later show in our results. The miner itself is distributed via a single JavaScript file, which the website's owner includes on the page along with a small snippet to configure and start the mining process. The snippet and its configuration may be further customized, e.g., to not execute on mobile devices, but at least requires a unique id that maps miners to identities. In the case of CoinHive these are called *site-keys* and are needed to account payouts for calculated hashes.

Under the hood, the miner is based on three browser technologies previously introduced in Section 2.2 when we discussed the trend of *emerging features*: WebSockets, WebWorkers, and WebAssembly. In the context of web-based mining, WebSockets allow the efficient communication between the individual miners and the mining pool that coordinates their efforts. However, WebSockets are also used in several other types of web applications, like chats and multiplayer games, and thus represent only a weak indicator of mining activity. While WebWorkers are not strictly necessary for implementing web-based mining, they allow for better utilizing the available resources by running the miner on multiple CPU cores in parallel and thus can also be found in most implementations. For our study, we hence consider the presence multiple of WebWorker threads as an indicator for potential mining activity. Last but not least, WebAssembly is a perfect match for implementing mining software, as it enables compiling cryptographic primitives, such as specific hash functions, from a high-level programming language to low-level code for a browser. Thus the calculation of the CryptoNight hashes at the core of the whole mining process is implemented in WebAssembly to increase performance, meaning that WebAssembly usage is another useful indicator for mining activity.

On startup, the CoinHive miner instantiates the desired number of WebWorkers, e.g., one worker per CPU core, and creates a WebSocket connection to the mining pool, where it registers itself with the supplied site-key. The corresponding worker code is usually included in the JavaScript code of the miner as a binary blob and instantiated by each worker. If throttling is configured to use less than 100% of the CPU for mining, the workers constantly monitor the time consumed for each calculated hash and adjust the delay between hash calculations. This, however, only allows for a rough approximation of the desired load on the system. While CoinHive allows a website's owner to conveniently set up a web-based miner without running a mining pool for themselves, the CoinHive service did keep a 30% cut of all mined currencies.

5.3 Scanning Methodology

Based on the discussed background, our goal is now to conduct a systematic study of WebAssembly usage on the Web. We obviously do not want to limit this study to only detect the known CoinHive miner, but also obfuscated variants and different mining implementations. For this, we need an automated method to collect all WebAssembly modules for later analyses and to detect indicators for mining activity, such as an unusual CPU utilization, the excessive repetition of functions and the presence of suspicious scripts. To achieve this, we again rely on a real browser controlled via the CDP as part of our security scanning pipeline. In the following, we describe our instrumentation for this use case, as well as our general approach to detect cryptojacking websites.

5.3.1 Collecting WebAssembly

First of all, we need a way to collect all WebAssembly modules that we discover while crawling the Web. Previous research mentions the undocumented `-dump-wasm-module` flag for Chrome's Wasm compiler as a simple option to save all executed Wasm modules [134]. However, the flag was only available in debug builds and later completely removed. Another option to collect all Wasm code is hooking the CDP's `Debugger.scriptParsed` event and filtering for the `wasm://` scheme. Though this event only gives us the code for each parsed function and not the whole Wasm module in its original form and hence important parts of the module, like the memory section, are not available. For comprehensiveness, we instead use monkey patching to transparently hook the creation of all JavaScript functions which can compile or instantiate Wasm modules. Figure 5.3 demonstrates how we hooked `instantiate` and the corresponding async version called `instantiateStreaming`. For `compile` and its async counterpart, the process is identical. For the `WebAssembly.Module` constructor, on the other hand, we use the built-in `Proxy` object to create a trap for the `new` operator [see 155].

```
let original = WebAssembly.instantiate;
WebAssembly.instantiate = function(bufferSource) {
    //Log bufferSource to backend here
    return original.call(WebAssembly, ...arguments);
};
WebAssembly.instantiateStreaming = async function(source, obj) {
    let response = await source;
    let body = await response.arrayBuffer();
    return WebAssembly.instantiate(body, obj);
};
```

Figure 5.3: Modifying the instantiation of WebAssembly to collect the raw Wasm bytes.

5.3.2 Detecting Cryptojacking

Compared to merely collecting all WebAssembly modules, accurately detecting which of these are used for web-based mining is a lot more involved. For this, we designed the dedicated cryptojacking detection process shown in Fig. 5.4 that spans three individual phases: 1) An over-permissive first broad sweep to identify potential miner *candidates* using heuristics, 2) a thorough run-time analysis to isolate the real miners within the candidate set, and 3) a generalization step, in which we extract static indicators, that allow the identification of non-active or stealth mining scripts.

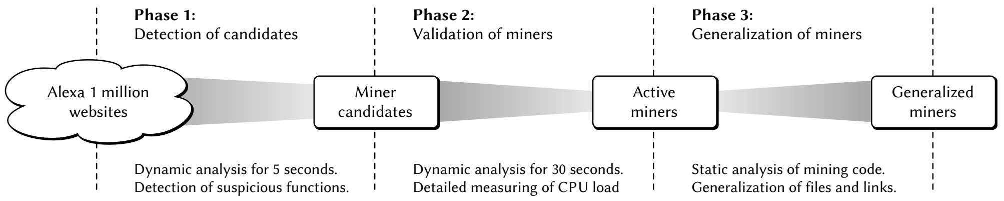
Phase 1:
Detection of candidates
Phase 2:
Validation of miners
Phase 3:
Generalization of miners
Alexa 1 million websites
Miner candidates
Active miners
Generalized miners
Dynamic analysis for 5 seconds.
Detection of suspicious functions.
Dynamic analysis for 30 seconds.
Detailed measuring of CPU load
Static analysis of mining code.
Generalization of files and links.

Figure 5.4: Overview of our approach for identification of web-based miners.

Phase 1: Detection of candidate sites In the first phase, our approach conducts a fast and imprecise initial analysis of websites to create a pool of candidates which likely—but not necessarily—host a mining script. To do so, we compiled a set of heuristics that hint the potential presence of a cryptojacking script and that can be measured at run-time while rendering a webpage in a browser. These heuristics were extracted from a manual analysis of verified mining scripts. For one, we initiate a short profiling of the site's CPU usage, with unusual high CPU utilization levels being interpreted as an indicator for mining. Furthermore, we mark all sites as suspicious, that use miner-typical web technologies, which are not in wide-spread use in the general web, namely WebAssembly or non-trivial amounts of WebWorkers. If at least one of these indicators could be found in a site, this site is marked as a potential *mining candidate*. Thus, the result of this phase is an over-approximation of the set of actual mining sites.

Phase 2: Validation of mining scripts For obvious reasons, none of the used heuristics is conclusive in the identification of miners, as there is a multitude of legitimate reasons to use WebWorkers, WebAssembly or temporary high amounts of computation. However, the constant and potentially unlimited usage of CPU, caused by a *single function* within parallelized scripts is a unique phenomenon of cryptojacking. Thus, in the second phase we conduct a significantly prolonged run-time analysis of the candidate sites, in which the sites receive *no* external interaction and hence should be idle after the initial rendering and set-up in legitimate cases. However, if once the page is loaded, all JavaScript is initialized, and the DOM is rendered, the CPU usage still remains on a high level and the computation load is the result of repetitive execution of a single function within the webpage's codebase, we conclude that the site hosts an *active* cryptojacking script.

Phase 3: Generalization of miner characteristics The run-time measurements of the first two phases limit our approach to the detection of *active* mining scripts. We thus might miss mining sites that are inactive at the time of the test, for instance due to programming errors in the site's JavaScript code, a delayed start of the mining operation, or mining scripts waiting for external events such as an initial user interaction with the page. To create a comprehensive overview, it is important to identify these sites to document the *intent of mining*. To this end, we leverage the results of the second phase to generate a set of static features of mining scripts which we can be used to reevaluate the data from phase 1.

To this end, we first extract the JavaScript code from the validated mining sites that is responsible for initiating and conducting the mining operations. From this script code, we take both the URL and a hash of its contents as two separate features. Furthermore, we collect all parsed WebAssembly functions, sort them and use the hash of the whole codebase as the third feature. We then apply each feature to our list of confirmed miners from the previous phase and keep only those that describe at least a certain number of miners. As the result, we obtain a set of generalized fingerprints, which can identify common mining scripts even in their *inactive* state. Applying these onto the data collected during the first phase in combination with our list of confirmed miners from the second phase yields to total number sites with a web-based miner.

5.3.3 Implementation Details

While we can not describe all implementation details of all three phases in detail, there are two specifics of our implementation that we want to nevertheless point out: how we deal with that fact that our measurement runs on one large server instead of a typical consumer hardware and how we can measure CPU caused by *individual functions* and not just the whole browser process.

Fake number of cores The number of logical cores of a visitor's CPU is exposed in JavaScript via the `hardwareConcurrency` property of the global `navigator` object. This allows scripts to adjust the number of concurrent WebWorkers according to the available hardware and is used by miners to start the desired number of threads (usually one per core). However, we do not want a single mining site to seize all available resources on our server and interfere with simultaneous visits of other websites. Furthermore, websites might employ checks on the number of cores and not run if an unusually high number is observed, thus preventing us from detecting them. Changing the returned value can be achieved by injecting a script into each document that overwrites the property before executing any other script content.

CPU Profiling

Most importantly, instead of using standard Unix tools to measure the CPU load on a *per-process* level, we utilize the integrated profiler of Chrome's underlying JavaScript engine V8 to measure the load on a *per-function* level. The profiler pauses the execution at a regular interval and samples the call stack, which enables us to estimate the time spent in each executed function. This way, we can not only determine if a *single function* consumes a considerable amount of CPU time, but also pinpoint the responsible script in the website's code by aggregating the collected data for each unique call stack. Wasm code itself cannot be profiled on a function level, so all samples of it are just named `<WASM UNNAMED>`. However, from the call stack we can still see how much time the Wasm code took and trace it back to the JavaScript function which caused the call into Wasm in the first place, as shown in Table 5.2. By comparing the time spent in a function with the length of the profiling, we can estimate the caused CPU load for that particular function. Note that if the same code is running in several workers simultaneously, the combined time from all workers can be as high as the number of cores times the length of the profiling, e.g., our profiling for 5,000 milliseconds with 4 CPU cores in phase 1 can result in a maximum time of 20,000. Thus, taking the value of 14,375 from Table 5.2 as an example, would mean this function generated a load of approximately 72%.

Table 5.2: Example of a call stack with the aggregated amount of samples and time spent for each of its functions.

Function name	# Samples	Time in ms
`<WASM UNNAMED>`	73,938	14,375.3
`Module._akki_hash`	1	0.1
`CryptonightWASMWrapper.hash`	4	0.6
`CryptonightWASMWrapper.workThrottled`	11	1.8
`(root)`	0	0.0

5.4 Large-Scale Study

As previously stated, our goal is to paint a comprehensive picture of the current WebAssembly practices in the wild. To this end, we measure the prevalence of cryptojacking, examine the effectiveness of the current generation of dedicated anti-cryptojacking countermeasures, and investigate what WebAssembly is used for beyond that.

5.4.1 Cryptojacking

We used the aforementioned browser instrumentation to automatically find instances of web-based cryptojacking in the wild. In what follows, we briefly discuss the key parameters of our experiment setup for each of the three phases. Then, we report on the prevalence of these attacks in the wild and investigate the effectiveness of existing defenses.

Experiment Setup We conducted *Phase 1* our study on the Alexa list of the top 1 million most popular sites. We visit the front page of each site and wait until the browser fires the load event or a maximum of 30 seconds pass. Furthermore, to allow for sites that dynamically load further content, we wait an additional 3 seconds or until no more network requests are pending. We then start the CPU profiler and measure all code execution for 5 seconds and flag the site as suspicious, if there is a function with more than 5% load on average. As the most CPU-heavy function on each website of the Alexa Top 1 million had an average load of only 0.2% $\pm$ 3.21, we reckon that a value of 5% or more warrants further investigation. We also flag the site for extended analysis, if either any Wasm code or more than 3 workers are used, which is equal or more than all the CPU cores we pretend to have. For this phase, we used a single server with 24 CPU cores and 32 GB of RAM running 24 simultaneous crawlers backed by Chrome v67.0.3396 over a time span of 4 days. The detailed verification of suspicious sites in *Phase 2*, we use the same general setup as the first phase. However, here we only run one crawler on a smaller server with 8 CPU cores. By visiting the websites one-by-one and profiling for a longer time of 30 seconds, we can more accurately determine if a website contains a mining script. If there is one function in the codebase that results in an average load of 10% or more, we label it as a confirmed and active miner. We argue that while a value lower than 10% certainly would make the miner very hard to detect, it also severely thwarts the ability to make money with cryptojacking. Furthermore, such slow mining does not even seem to be supported by popular mining scripts, as we will describe shortly. In the final *Phase 3*, we create the fingerprints as previously outlined using the code of the confirmed miners. However, we only keep the fingerprints shared by at least 1% of all miners. This restrictive measure ensures that only mining scripts with multiple validated instances produce fingerprints and, thus, avoids accidental inflation of potential phase 2 classification mistakes. The resulting fingerprints are then applied to the collected data from the first phase, yielding the final number of websites employing cryptojacking on their visitors.

To validate that our implementation and setup are working as intended, we created a testbed with the two popular implementations that start without the user's consent: CoinHive and CryptoLoot. This testbed consists of 24 locally hosted pages, which each contain one of the miners at a different throttling levels between 0% and 99%. Interestingly, even if the miner is configured with a throttle as high as 99%, so that it should utilize only 1% CPU, we can confirm it as a miner with our 10% threshold. Looking into the implementation of the throttling, we find that the code never sleeps for longer than two seconds between hash calculations, which makes it impossible to actually use very low throttling values. We also confirm this by monitoring Chrome's CPU usage with `htop` and find that no matter how high we set the throttling, the load on our machine never drops to below 20%. As our implementation is able to successfully detect all miners in the testbed, regardless of the used throttling value, we are confident its ability to find active cryptojacking scripts.

Prevalence As first result of our crawling, we identify 4,627 suspicious sites in the Alexa ranking using the methodology and parameters outlined in the previous sections. Out of these, 3,028 are flagged for having a load-intensive function, 3,561 for using at least as many workers as CPU cores we pretend to have and 2,477 for using Wasm. Note that these sets overlap, as for example the usage of Wasm often implies a CPU intensive application. The detailed analysis of these 4,627 suspicious sites results in 1,939 sites with a continuously high CPU usage over a profiling for 30 seconds. We use the resulting set of scripts for the third phase to build fingerprints of the most popular miners, resulting in 15 hashes of JavaScript code, 12 hashes of Wasm codebases, and 8 script URLs. The latter can be found in Table 5.4. After applying these fingerprints, we obtain the final number of 2,506 websites, which are very likely to employ cryptojacking. Table 5.3 summarizes our results.

Table 5.3: Prevalence of miners in the Alexa Top 1 million.

Phase	Result	# Websites	% of Alexa
1	Suspicious sites	4,627	0.46
2	Active cryptojacking sites	1,939	0.19
3	Total cryptojacking sites	2,506	0.25

Table 5.4: Common script URLs responsible for the creation of the mining workers, which resulted in fingerprints.

URL	# Occurrences
`//coinhive.com/lib/coinhive.min.js`	656
`//advisorstat.space/js/algorithms/advisor.wasm.js`	311
`//www.weather.gr/scripts/ayh9.js`	68
`//aster18cdn.nl/bootstrap.min.js`	59
`//cryptaloot.pro/lib/crypta.js`	46
`//gninimorenom.fi/sytytystulppa.js`	35
`//coinpot.co/js/mine`	27
`//mepirtedic.com/amo.js`	22

During manual investigation of a sample of the additional 567 sites only detected in phase 3, we found five reasons why our dynamic analysis missed these miners: (1) A script for web-based mining is included, but the miner is never started. (2) The miner only starts once the user interacts with the web page or after a certain delay. (3) The miner is broken—either because of invalid modifications or because the remote API has changed. (4) The WebSocket backend is not responding, which prevents the miner from running. (5) The miner is only present during some visits, e.g., to hinder detection or due to ad banner rotation. This analysis confirms the need for a three-step identification process, as only the combination of phase 2 and 3 enable us to determine a comprehensive picture of current cryptojacking in the websites of the Alexa ranking.

Effectiveness of Countermeasures To both compare our findings to existing approaches for the detection of cryptojacking and to validate our results, we select three popular solutions to block miners in the browser. For one, we use the *NoCoin adblock list* [186], which is a generic list for adblockers, such as Adblock Plus or uBlock Origin and is now also used by Opera's built in adblocker. For the remainder of this section, we refer to this list as *Adblocker*. Furthermore, we include the blacklists used by the two most popular Chrome extensions with the purpose of blocking web-based miners: *No Coin* [187] and *MinerBlock* [162]. We extract the detection rules these extensions contain and translate them into SQL statements while preserving the wildcards, in order to apply them to the data collected during our crawl of the Alexa Top 1 million sites. The number of identified miners for each system are presented in Table 5.5 in the first column. The other columns of this table compare these results for each system with the 2,506 websites we identified as miners. The second column reports on the intersection of both lists, that is the number of sites on which both approaches are in agreement. Accordingly, the last two columns each contain the number of sites that one approach reported, but not the other.

Table 5.5: Detection results of our approach and three common blacklists as absolute numbers.

Blacklist	# Detections	# Both	# Only they	# Only we
Minerblock	1,599	1,402	197	1,104
No Coin	1,217	1,039	178	1,467
Adblocker	1,136	1,049	87	1,457

Unsurprisingly, our approach mixing static and dynamic analysis clearly outperforms the three static blacklists and spots a considerable amount of additional web-based miners. Moreover, the large overlap in sites that both we and the extensions found, validates that our approach and shows that it is indeed suitable to detect cryptojacking in the wild. There are, however, a few sites that our approach misses, but the blacklists detect. Manual analysis of a subset showed that besides overly zealous lists, the main reason is that we can only learn fingerprints of *active* miners. For example, some website owners copied CoinHive's script to host it on their own servers a few months ago. Meanwhile, all these mining scripts stopped working, as CoinHive changed its API used in the communication with the pool. Therefore, while this probably represents a cluster of inactive miners, we are unable to detect them, as no fingerprint for any of the scripts could be generated in the third phase, due to the fact that the *whole* cluster was inactive at the time of analysis.

The existing blacklists on the other hand can detect them, as their rules are curated by humans, which allows them to apply a couple of generic measures. For example, most blacklists include a rule for `*/coinhive.min.js`. In contrast, our static indicators are generated in a fully automated fashion, based on code characteristics from dynamically validated miner instances. In this process, we cannot generalize our list of fingerprinted full script URLs towards partial URLs or even only filenames without manual review, as this could lead to misclassifications. For instance, in our dataset such an attempt would end up in all scripts named `*/bootstrap.min.js` being blacklisted because a widespread mining script uses this benign-sounding name, as shown in Table 5.4.

5.4.2 WebAssembly Ecosystem

In a follow-up study later in the same year, we set out investigate the larger WebAssembly ecosystem, without limiting our efforts to cryptojacking only. We conducted a similar crawl of the Alexa Top 1 million and discovered 1639 sites loading 1950 Wasm modules. This is a significant decrease compared to the 2,477 sites using WebAssembly during our first study. We attribute this to the fact that the price of Monero in USD dropped to about a fifth of its value between those to studies, thus severely limiting profits [48].

WebAssembly modules Of the 1950 collected modules, 150 are unique samples. This means that some Wasm modules are popular enough to be found on many different sites, in one case the exact same module was present on 346 different sites. On the other hand, 87 samples are completely unique and were found only on one site, which indicates that many modules are a custom development for one website. On some pages, we found up to 10 different Wasm modules, with an average of 1.22 modules per page on sites that use WebAssembly at least once. Moreover, Fig. 5.5 shows that the more popular sites also tend to use WebAssembly more often.

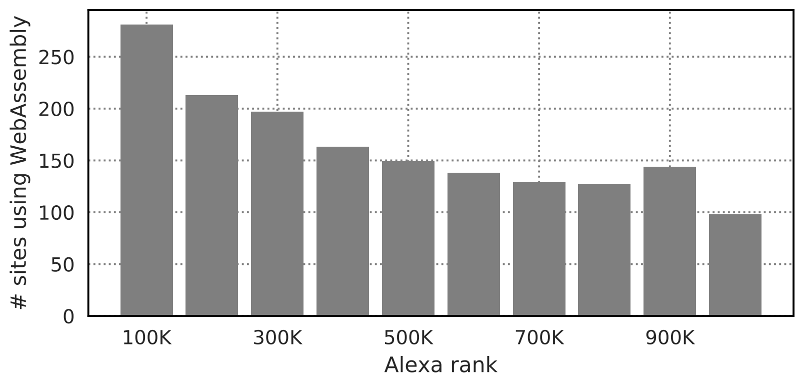

Figure 5.5: Distribution of Alexa sites using WebAssembly in bins of 100,000 sites.

Regarding the initiator of the modules, on 1118 sites the module was instantiated by a first-party script, while on 795 sites the module came from a third-party script or iframe with another origin. In the second case, the site's administrator might not even be aware that WebAssembly is used on his/her site. Note that there is some overlap, as some sites used multiple modules, especially since we also crawled several subpages for each site. The mere instantiation of a Wasm module, however, does not mean that a site is actively using the module's code. Some sites just test if the browser does support WebAssembly, while other sites are actually relying on the functionality the module exposes. A first indicator for this is the size of the module: the smallest was only 8 bytes, while the largest was 25.3 MB with a median value of 99.7 KB per module, as can be seen in Figure 5.6. On the other hand, the JavaScript code on the sites using Wasm had a median size of 2.79 MB. This shows that currently the amount of Wasm code is often only a fraction of the total codebase on a site. Nevertheless, this should be seen only as a rough estimate as the comparison between a text-based and a binary format is inherently unfair.

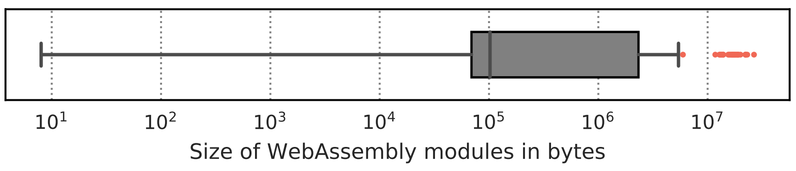

Figure 5.6: Distribution of the size of WebAssembly modules. The box represents the middle 50% of the data, with the line in the box at the median. Outliers are shown as red dots.

Manual analysis To find out what they are actually used for, we manually analyzed all 150 collected Wasm modules. We first inspected the modules themselves and looked at function names and embedded strings to get an idea about the likely purpose of the module. In order to confirm our assumptions, we visited one or more websites that used the module and investigated where the Wasm module is loaded and how it interacts with the site. Thereby, we arrived at the following six categories: Custom, Game, Library, Mining, Obfuscation, and Test. The first three are of benign nature, but modules of the categories *Mining* and *Obfuscation* use WebAssembly with malicious intentions. We consider testing for WebAssembly support in general as neither benign nor malicious itself and thus see the category *Test* as neutral.

Unsurprisingly, the largest observed category implements a cryptocurrency miner in WebAssembly, for which we found 48 unique samples on 913 sites in the Alexa Top 1 Million. With 44 samples we found almost as much different games using Wasm, but in contrast to the miners, these games are spread over only 58 sites and thus often only appeared once. For 4 modules we could not determine their purpose and labeled them as *Unknown*. Of these, 2 did not contain a valid Wasm module, but the sites attempted to load it as such regardless. Table 5.6 summarizes our results and shows that with 56%, the majority of all WebAssembly usage in the Alexa Top 1 Million is for malicious purposes. In the following, we will briefly discuss each category except the web-based miners, as they were already covered in detail in our previous study on the cryptojacking phenomenon.

Table 5.6: The prevalence of each use case in the Alexa Top 1 Mio. As some sites had modules from multiple categories, the sum exceeds 100%.

Category	# of unique samples		# of websites		Malicious
Custom	17	(11.3%)	14	(0.9%)	
Game	44	(29.3%)	58	(3.5%)	
Library	25	(16.7%)	636	(38.8%)	
Mining	48	(32.0%)	913	(55.7%)	✗
Obfuscation	10	(6.7%)	4	(0.2%)	✗
Test	2	(1.3%)	244	(14.9%)	
Unknown	4	(2.7%)	5	(0.3%)	
Total	150	(100.0%)	1,639	(100.0%)	

Benign: Custom, Games and Libraries Modules in the *Custom* category appeared to be a one-of-a-kind experiment, e.g. one was a fancy background animation and another collected module contained an attempt to write a site mostly in C# with cross-compilation to Wasm. *Games* are arguably also of custom nature and often only found on one specific site. However, they are a very specific subset of custom code, of more professional nature and often also have a clear business model (e.g. in-game purchases or advertisements). *Library*, on the other hand, describes Wasm modules that are part of publicly available JavaScript libraries. For example, the library *Hyphenopoly* uses WebAssembly under the hood to speed up its algorithm for word hyphenation. In this case, the use of WebAssembly might not be the result of an active decision by the site's developer.

Neutral: Test As mentioned in the beginning, some sites loaded WebAssembly modules with a size of only a few bytes. Manual investigation showed that the only purpose of these modules is to test whether WebAssembly is supported by the visitor's browser. We discovered such test modules on 244 sites. Of these, 231 sites then proceeded to load another module after this test, while, on the other hand, 13 sites only made the test without executing any further Wasm code afterward. The latter might, for example, use this in an attempt to fingerprint their visitors. However, due to the lack of information we gain from such a small module, we see these as neither benign nor malicious.

Malicious: Obfuscation While cryptojacking certainly did get a lot of attention from the academic community in 2018 [e.g. 134, 109, 214, 218], this is not the only type of malicious WebAssembly code already in use in the wild. Rather than using WebAssembly for its performance improvements, some actors abuse its novelty instead. Fig. 5.7 shows the HTML and JavaScript code embedded into the memory section of a Wasm module we found. Through this obfuscation of hiding the JavaScript code in a Wasm module, malicious actors likely can prevent detection by analysis tools that only support JavaScript and do not understand the Wasm format. The code tries to create a pop-under, which is an advertisement that spawns a new window behind the current one and is basically the opposite of a pop-up. The idea is that this way, the window stays open for a very long time in the background until the user minimizes or closes the active browser window in the foreground. Another 8 modules also contained code related to popups and tracking in the memory section, likely in an attempt to circumvent adblockers. The last of the 10 modules, which employed obfuscations via WebAssembly, implemented a simple XOR decryption (and nothing else). This could, for example, be used to decrypt the rest of a malicious payload. However, in this case, the module seemed to not be used at all after the initialization. Nevertheless, we see these first, simple examples as evidence that malicious actors are already slowly moving towards WebAssembly for their misdeeds and we expect more sophisticated malware incorporating Wasm code to emerge in the future.

```
<script>
    var popunder = {expire: 12,
    url: '//hook-ups-here.com/?u=813pd0x&o=4gwkpzn&t=all'};
</script>
<script src='//hook-ups-here.com/js/popunder.js'></script>
```

Figure 5.7: Code to create a pop-under advertisement, which was found in the memory section of a WebAssembly module.

5.5 Related Work

To the best of our knowledge, at the time of publishing these results in 2019, there were no peer-reviewed publications on the general security impact of WebAssembly modules on the Web. There were, however, already some tools in the work by academics, e.g., a dynamic analysis framework for WebAssembly called Wasabi [268].

For the cryptojacking phenomenon, on the other hand, various related works already existed: The study by Eskandari et al. [72] was the first to provide a peek at the problem. However, the study is limited to vanilla CoinHive miners, and the underlying methodology is unsuited to detect alternative or obfuscated mining scripts. More recently, Konoth et al. [134] searched the web for instances of drive-by mining and proposed a novel detection based on the identification of cryptographic primitives inside the Wasm code. Similarly, Wang et al. [267] detect miners by observing bytecode instruction counts, while Rodriguez and Posegga [214] use API monitors and machine learning. Our work, on the other hand, uses a sampling profiler to detect busy functions and is thus more closely related to the work by Hong et al. [109]. However, we crawled the whole Alexa Top 1 Million, while their study was limited to the Top 100k. Furthermore, fluctuations in the Alexa lists and the short timespan of mining campaigns add uncertainty to previously presented results. Therefore, our study provides an additional, independent data point on this new phenomenon at a different point in time. These factors are also the reason why we decided against directly comparing the number of detections between the papers.

Furthermore, unauthorized mining of cryptocurrencies is not limited to web scenarios. For example, Huang et al. [111] present a study on malware families and botnets that use Bitcoin mining on compromised computers. Similarly, Ali et al. [6] investigate botnets that mine alternative currencies, such as Dogecoin, due to the rising difficulty of profitably generating Bitcoins. To detect illegitimate mining activities, either through compromised machines or malicious users, Tahir et al. [247] propose *MineGuard*, a hypervisor-based tool that identifies mining operations through CPU and GPU monitoring. Our study extends this body of work by providing an in-depth view of mining activity in the web.

From a more general point of view, cryptocurrency mining is a form of *parasitic computing*, a type of attack first proposed by Barabási et al. [14]. As an example of this attack, the authors present a sophisticated scheme that tricks network nodes into solving computational problems by engaging them in standard communication. Moreover, Rodriguez and Posegga [215] present an alternative method for abusing web technology that enables building a rogue storage network. Unlike cryptojacking, these attack scenarios are mainly of theoretical nature, and the authors do not provide evidence of any occurrence in the wild. On a technical level, our methodology is related to approaches using high-interaction honey browsers [e.g., 207, 266, 166, 133], which are mainly utilized to detect attacks on the browser's host system via the exploitation of memory corruption flaws, a threat also known as *drive-by-downloads*. While our approach shares the same exploration mechanism — using a browser-like system to actively visit potentially malicious sites — our detection approach diverges, as the symptoms of browser-based mining stem from the exclusive usage of legitimate functionality, in contrast to drive-by-download attacks that cause low-level control-flow changes in the attacked browser or host system.

6 Revealing Insecure Automated Browsers

The previous three chapters showed how our modern security scanner based on browser instrumentation can be adapted for many different uses cases and can overcome challenges like a blurring of involved parties and the use of emergent technologies that traditional scanners struggle with. Unfortunately, the approach of using a real browser in an automated tool also has a significant drawback: the huge codebase of a browser also results in a much larger attack surface compared to simpler tools that, e.g., only handle the HTTP communication without executing active content. This means exposing an automated browser as a part of a tool can be problematic, if it visits potentially untrusted URLs. In this chapter, we will search for such *server-side browsers* and analyze how many of them are using outdated versions with known critical vulnerabilities. While we will still use our scanning pipeline to discover these vulnerable browsers on a large scale, this should not be considered as yet another use case for our scanner. Instead, the focus of this chapter to highlight the potential problems of using a real browser as part of an automated system, regardless of whether it is part of a security scanner or another product.

In Section 6.1, we first introduce our terminology and outline some use-cases that require a real browser to visit user-provided and thus untrusted URLs. Then we describe how our threat model compares to traditional servers-side request forgery attacks, which share some similarities, and describe the exact scope of our study. In Section 6.3, we outline the methodology that we use to discover as many server-side requests as possible on a large scale. For this, we present three approaches that can entice servers that we crawl to visit our servers in response. After that, in Section 6.2, we describe our approach to detect bots using real browsers in the incoming traffic, thus filtering out both traffic by human users and by simpler request tools. We use this methodology of enticing requests and identifying server-side browsers to conduct a large-scale study of vulnerable automated browsers in the wild and comprehensively report on our findings in Section 6.4. Finally, in Section 6.5, we discuss related works from the general area of browser fingerprinting and bot detection, as well as works on the specific topic of vulnerabilities caused by server-side requests.

6.1 Server-Side Browsers

Nowadays, many applications that were originally intended as ordinary desktop software, such as messengers or word processors, are moving to the Web. As a result, new technologies are implemented to support this transition. One of these are *server-side requests* (SSRs), i.e., when the web server also acts as a client to fetch additional information from other locations on the Web. Such SSR implementations can often be triggered by unprivileged users of a website, e.g., social networks showing a preview for all user-submitted links. Traditional SSR implementations resemble tools like *wget* or *curl* and only fetch the main HTML document in a single request without executing active content. However, the rise of JavaScript-heavy frameworks and single page applications (SPAs) made this approach largely incompatible with modern web development, as the relevant content is often dynamically inserted afterward. Consequently, this led to an increased usage of what we call *server-side browsers* (SSBs), i.e., real browser like *Headless Chrome* running on the server-side. By using such an SSB instead of a simple SSR, the back end can not only send HTTP requests and receive the responses but render entire web pages built with modern JavaScript frameworks. However, running a real browser that *anyone* can summon to an *arbitrary* URL naturally poses a much higher risk than downloading a page without executing its code, as we investigate in this chapter.

6.1.1 Terminology and Use Cases

In this section, we first introduce our terminology and then describe multiple use cases for server-side browsers that can not be solved with simpler alternatives, showing their usage is often intended and unavoidable. After that, we discuss the potential security pitfalls posed by this technology.

Terminology Basically, SSRs are conducted each time one server requests data from another, e.g., from a service offering an HTTP API. These SSR implementations are characterized by the fact that they only handle the connection on the HTTP level, but treat the returned content as a string or simple data format like JSON. In particular, they do not parse and render HTML, do not load embedded resources, and do not execute active content like JavaScript code. Popular SSR implementations include command-line tools like *curl* and *wget*, as well as the *http* package for Node.js, and the *file_get_contents* function for PHP. SSBs, on the other hand, are the equivalent of running a normal desktop browser on the server-side. The obvious difference here is that there is no human interacting with a visible graphical user interface involved. Instead, these SSBs are fully automated tools running in the background. Under the hood, they support exactly the same features as their desktop equivalent, in particular, they parse and execute all active JavaScript content unless this is explicitly disabled. Popular SSB implementations rely on tools like *Headless Chrome* and *Puppeteer*, which were already introduced in Section 2.3 when discussed the various browser instrumentation approaches. Please note that our terminology implies that each SSB also conducts one or more SSRs while loading a webpage, but not each SSR necessarily comes from an SSB.

Use cases There are various use cases that require running a fully-featured browser on the server-side and can not be solved by resorting to simpler, SSR-like solutions. For example, major search engines such as *Google* and *Bing* rely on SSBs for their web crawlers [92, 159]. In principle, statically crawling a web page without rendering it would be more convenient as it uses a lot less resources. Yet, as many of today's web applications are built dynamically with JavaScript, web crawlers need to be capable of executing this scripting language to properly index the page's content. Therefore, using an SSB with a native rendering engine increases the accuracy of the indexed content, as SPAs might only show a blank page if JavaScript is disabled. Another use case are web security services like *Symantec Sitereview* [245] and *VirusTotal* [260] offering ratings and categorizations for URLs, allowing users to check whether a website is potentially malicious. As previous work finds, a significant percentage of search results and ads employ cloaking techniques [265, 264, 145] which involve blacklisting of IPs and user agents affiliated with search engines or detecting the absence of JavaScript execution. This way, they try to hide from these services by serving harmless content to web crawlers while serving malicious sites to potential victims [117]. These techniques make it especially important for providers of web security scanners to use real, automated browsers to mimic a human user in order to scan websites from a human's and not from a bot's perspective.

6.1.2 Threat Model and Scope

As long as the URL in an SSR/SSB implementation is hard-coded to visit only one trustworthy server, running them is generally unproblematic. Problems arise if this URL depends on user input, e.g., in all the previously outlined use cases. An apparent threat in this scenario are *server-side request forgery* (SSRF) attacks. In the most common SSRF scenario, an attacker tries to bypass firewall rules to gain access to privileged parts of the network. In the simplest case, submitting an internal IP address such as `http://10.0.0.1` to a vulnerable SSR service would result in that server forwarding the internal content to the external attacker. Other, similar attacks against SSRs are abusing them as an attack proxy, e.g., in a denial of service (DoS) attack, or abusing them to confuse client-side filters integrated into browsers in a so-called *origin laundering attack*. The high prevalence of SSRs as a convenience feature, as well as its increasing severity due to complex architectures and higher adoption rates of cloud and web services has earned SSRF a place in this year's OWASP Top 10 [193]. These scenarios and their potential consequences were studied in detail in a paper on SSRF attacks published in 2016 by Pellegrino et al. [198].

On the other hand, one so far over-looked scenario is directly attacking the implementation of the requesting mechanism, i.e., the headless browser itself. As these SSBs are based on a real browser and visit arbitrary, attacker-supplied URLs, they too are vulnerable to JavaScript exploits like every other desktop browser. Successful exploitation means the attacker could gain control of the server the SSB is running on. In this scenario, there is no request forged and no deputy confused, instead the attacker uses the request service *as intended* and prompts it to visits an *external* website under the attacker's control. From there, they can launch their JavaScript payload and probe the visiting user agent for exploitable vulnerabilities. If successful, they might be able to completely compromise the server and begin lateral movement through the network from there.

A major threat to all browsers is the large number of publicly disclosed vulnerabilities. Google Chrome, for example, enjoys a rapid update cycle to keep up with security and new features of the constantly evolving web ecosystem. A major update of the stable channel is pushed to the public every six weeks [84] while minor releases come every two to three weeks according to Chrome's update strategy [87]. Google even announced plans to further increase the frequency of their updates and plan to release a major update every four weeks by the end of 2021 [40]. They aim to further reduce the *patch gap*, i.e., the time between a security bug fix being merged into their open-source repository and the release of a new, stable version that includes this bug fix for all users [91].

Yet one main difference between headless browsers and their desktop equivalent is the way updates are handled: While desktop browsers usually update automatically nowadays, headless browsers require manual intervention to keep up with security patches. Not updating these SSBs automatically has good reason, since the automation API or other important features might have changed and thus could silently break tools that rely on them. For example, Google writes in the official Puppeteer repository: "We see Puppeteer as an *indivisible entity* with Chromium. Each version of Puppeteer bundles a specific version of Chromium – the *only version* it is guaranteed to work with. [...] This is not an artificial constraint." [85]. This means there is a significant risk of outdated versions of SSBs still running in the wild if they are not constantly monitored and maintained. In this chapter, we thus focus on *automated but outdated browsers* that we can lure to visit a site under our control to (theoretically) deliver a publicly available JavaScript exploit from there. To summarize, we consider the following to be in and out of scope for this work respectively:

In scope All automated, server-side browsers that run a real rendering engine and execute JavaScript, regardless of which automation framework or headless browser implementation they use.

Out of scope All SSR implementations that do not use a real browser (e.g., curl) or use one, but have disabled JavaScript execution. Also, all SSRF attacks like accessing the internal network, bypassing URL filters, or origin laundering.

6.2 Identifying Vulnerable SSBs

In this section, we assume we already have some way to entice automated browsers to visit a server under our control, i.e., a monitoring server that collects data for all incoming requests. For the collected data to be usable, we first need to make sure that the requests are actually coming from automated systems and not human users. Moreover, we need a way to accurately determine which of these incoming requests are coming from *vulnerable* browsers. The most conclusive way to test if the connecting browsers are vulnerable would be to use real JavaScript exploits and test which of them successfully compromises the visitors. Clearly, this would be neither legal nor ethical and is not an option. Instead, we resort to safe fingerprinting techniques that can determine the most likely user agent of our visitors and then map this information to a list of known vulnerabilities.

6.2.1 JavaScript Metadata

First of all, our server records some basic information like the IP address, `User-Agent` header, and timestamp for each incoming request. To distinguish simple SSRs from real browsers, we reply with an HTML page that contains one small inline script to all incoming requests. If this script executes, it sends a notification to our monitoring server. This way, we can not only discern visitors that execute JavaScript from those who do not but also collect additional metadata and send it to our back end. We specifically designed this website to also work with very old browsers, by not using any HTML5 features and writing our inline script according to the ECMAScript 5 standard, which was released in 2009 [65]. This website also does not include any external resources to load with a single request and the inline script executes in about 5 ms, to make sure that the notification is sent to our server before the page is closed again.

6.2.2 Bots vs. Human Visitors

While discerning SSR tools with and without JavaScript is straight-forward with our monitoring website, discerning bots from humans, on the other hand, is a much more difficult task and often solved by either behavioral observation [120] or with CAPTCHAs [11]. Observing visitors over a series of requests allows finding indicators for bot-like behavior. We, however, do not run a real website with content that we want to protect from scraping, and instead need to discern humans from bots within *a single request* to our monitoring back end, making behavioral analysis not an option. Using a CAPTCHA is similarly not applicable, because every human visitor to our website that is asked to solve one, but instead closes the tab, would be misclassified as a bot. Previous work has shown that no programmatic bot indicator holds on its own and that even commercial fingerprinting companies lack robustness against concealed crawlers [258]. However, we have the unique advantage of knowing *when* to expect incoming bot traffic, because we first supply unique URLs to other websites in our attempts to trigger visits to our monitoring server, as we will describe in the next section. For this reason, our bot detection is based on the time period that passed between our initial request and the received SSR. We choose a very narrow time frame of 3 minutes starting from our initial trigger, because chances are extremely high that requests in that period come from a completely automated system. Even the most zealous administrator is unlikely to respond *that fast* to a new URL submitted to their website. Moreover, we extend this timing-based approach with additional bot indicators as described in the following.

6.2.3 Additional Bot Indicators

To better incorporate slower bots, we add several additional bot indicators and increase our timing threshold accordingly. For this, we use the following insight: even though bots might try to conceal themselves as human visitors, hardly any human visitor will try to mimic an automated browser. Therefore, while the absence of bot properties is rather meaningless, their presence presents—to some extent—a useful indicator. For this, we use the following four bot indicators: known crawler user agents, inconsistent user agents, inconsistent screen dimensions and missing plugin information. As this part was contributed by Robin Kirchner, we omit further details about the indicators here and refer the reader to our publication [176]. For each of these four indicators that is present during the visit, we double the initial threshold of 3 minutes during which we consider requests to be originating from bots. Therefore, if three of the four additional indicators would be present, the request would be labeled as bot traffic if it arrives within the first 24 minutes after our visit to their page. To summarize, we consider every visitor to be a human, until we find some hard indicators that they have to be a bot, e.g., because they are too fast and provide inconstant information about themselves. While this approach still overlooks slower bots, as well as particularly stealthy bots, we get a very reliable data set with little to no false positives, as it is very unlikely that human visitors have these indicators present and even if they do, it is even more unlikely that they are also fast enough for us to flag them as bots.

6.2.4 Feature Fingerprinting

Now that we know which visitors are bots running a server-side browser, we need to determine their browser version in order to identify those that are based on outdated browsers. However, as already described, the user agent string is trivial to spoof and sometimes even the user agent provided in the HTTP header and the user agent according to `navigator.platform` do not match—a clear indication of intentional manipulations. Therefore, the inline script on our monitoring website conducts a JavaScript *feature fingerprinting* that tests which features are supported by the visiting browser, to estimate the most likely *actual* user agent in a more reliable fashion. For this fingerprinting, we first probe for a few specific features to distinguish between the different browser *products*. For example, the presence of `InstallTrigger` is a unique indicator for the Firefox browser [234]. Now, to also distinguish between the different browser *versions*, we need a more sophisticated approach that works as follows: First, we compile a list of all JavaScript objects and properties of the global `window` object once, using the latest alpha release of Google Chrome. It is important to note, that we compiled this list while visiting an *HTTP* origin, as some newer features such as sensors are only available on a *HTTPS* origin. While our monitoring website is served on both protocols, for this fingerprinting we only rely on features that are always available, regardless of the protocol. At the time of writing, this list was compiled using Chrome 89 and contains 590 entries, such as `AggregateError`, `SVGAngle`, `Uint8Array`, and `WebGLQuery`. We embed this whole list into our inline script so that for each visitor with JavaScript enabled the presence of each of the 590 features is tested. We then encode the presence or absence of each feature into a long binary string and send this *feature vector* to our monitoring back end, along with the previously described other metadata.

6.2.5 Resulting User Agent

In our back end, we now need to decode this collected feature vector and map it back to the most likely browser versions. For this, we make use of the raw compatibility data provided by Mozilla's MDN [170], which is available on GitHub in their *browser-compat-data* repository [169]. As an example, their data shows that the `AggregateError` object is available in Chrome and Edge since v85, in Firefox since v79, in Safari since v14, and not supported by Opera. With this information for each of the 590 different features, we can narrow down the most likely browser version for each individual feature string. As the examples in Table 6.1 show, even with only a handful of features the range of possible versions can be quite small. For example, if a visitor supports `WeakRef` but not `AggregateError` then they are using Chrome, as Firefox introduced both features in the same update and all other browsers do not support `WeakRef` yet. Moreover, as `WeakRef` was introduced in Chrome 84 and `AggregateError` in Chrome 85, we now even know their exact major version for certain. However, it should be noted that not all fingerprint vectors are as distinct as this example and sometimes two successive releases might be indistinguishable.

Table 6.1: Five selected features and since which release the different browsers support them. The combination of presence (✓) and absence (✗) of the different features concludes that example 1 must be exactly Chrome 84 and example 2 must be Firefox 79 or newer.

Feature	Supported since				Example	
	Chrome	Firefox	Opera	Safari	1	2
AggregateError	85	79	—	14	✗	✓
MutationObserver	26	14	15	7	✓	✓
RTCCertificate	49	42	36	12	✓	✓
TrustedScript	83	—	69	—	✓	✗
WeakRef	84	79	—	—	✓	✓

This means we now have three potentially different user agents: from the HTTP headers, from the JavaScript navigator object, and from our feature fingerprinting. If these three agree with each other, then there is a high chance that the provided user agent information is indeed correct. When they do not match with each other, we could in doubt rely on the fingerprinting results as it is the most difficult to spoof. Yet, there is a chance that some administrators intentionally disabled a few features as an additional security measure. Therefore, we instead use the user agent with the highest browser version of the three as a conservative estimate, which in doubt defaults to the most secure of the three.

6.3 Scanning Methodology

In this section, we describe the other end of our setup, i.e., how we can entice web applications to conduct server-side requests to our monitoring server. As SSRs are implemented in the back end of a web application, their presence is not immediately obvious by accessing the front end, i.e., the loaded website with HTML, CSS, and JS content. Therefore, we require an empirical method to interact with these websites and reveal any potential SSRs. This means we need to modify our web security scanner to provide these websites with unique URLs pointing to our monitoring server. If we afterward observe a request to the URL we provided, there is a good chance that we successfully triggered an SSR. In general, there are three ways to entice the targeted server to visit us: First, we can supply *additional headers* that contain our URLs in each of our requests, e.g., in the *Referer* header. Second, we can modify existing *URL parameters* in GET requests that contain encoded URLs pointing to other domains and instead make them point at our domain. Third, we can submit our URLs in the *HTTP body* of a POST requests, e.g., by filling out HTML forms on the pages we visit. In the following, we will describe each approach in more detail.

6.3.1 HTTP Headers

HTTP headers offer a convenient option for transferring metadata to a web server. Usually, they serve various purposes such as transmitting user agent strings, language preferences, or cookies with session information. However, there are also some lesser-known headers like *True-Client-IP* or *Forwarded* that usually contain hostnames or IPs and can cause certain systems like reverse proxies or load balancers to reveal themselves and send requests to these remote hosts [203]. In a preliminary study, we investigated the impact of 25 different headers taken from the *Collaborator Everywhere* project [204], which is a plugin for the *Burp Suite* proxy [202] that injects specific headers of this type into every intercepted HTTP request. During our crawls with these additional headers, we found that with 98%, almost all SSRs triggered by headers were caused by supplying a URL in the Referer header. On the other hand, some of these other 24 headers decreased the amount of successfully visited websites by up to 28%. Due to these results, we only use the Referer header and do not send other headers to discover SSR implementations.

6.3.2 URL Parameters

To discover parameters that could trigger an SSR, we scan the URL query string of all HTTP requests to all the included resources that were requested while loading a page. If the query string of one of these requests contains one or more (potentially encoded) URLs, we mark its position as a potential SSR candidate. Moreover, we also search existing query keys for interesting names like *url*, *host*, or *domain* and mark their value as an SSR candidate, regardless of if their value contained a URL or not. We then replace each SSR candidate separately with a unique subdomain pointing to our monitoring server and request the resulting URL again. Fig. 6.1 contains our complete list of 22 SSR candidate names and is based on previous research on the detection of SSRF vulnerabilities and bug bounty reports [204, 30, 118, 59]. Fig. 6.2 shows one example of how a URL might get modified in this process.

```
action, addr, address, domain, from, host, href, http_host, load, page, preview,
↪ proxy, ref, referer, referrer, rref, site, src, target, url, uri, web
```

Figure 6.1: The full list of 22 additional SSR URL parameter candidate names

It is important to note that only one parameter is modified at a time while the other remains unchanged, to increase the likelihood of the back end successfully processing our request. If our monitoring server then receives an incoming HTTP request on the respective unique subdomain, we can trace it back to the initiating parameter. Compared to the previously presented approach of inserting additional HTTP headers, this procedure thus requires multiple requests to the same resource—one for each SSR candidate.

```
http://example.com/some/service?param=foo&id=42
↪  &url=http%3A%2F%2Fsome-other-service.example.org&exec=yes

http://example.com/some/service?param=foo&id=42
↪  &url=http%3A%2F%2Funique-subdomain.our.monitoring&exec=yes
```

Figure 6.2: One example of an URL before and after our replacement of SSR candidates

6.3.3 HTML Forms

HTML forms often accept URLs in one or more fields and thus present another opportunity to discover SSRs. However, unlike URL parameters that can be collected by simply visiting websites, these POST requests usually have to be triggered by user interaction. Moreover, many forms implement both client- and server-side validation mechanisms designed to reject any input entered in the wrong format. Hence, to submit a form we need to fill each field with the correct input of the expected data type while inserting as many SSR–URLs as possible. For this, we extended our security scanner with a form-filling algorithm, which tries to satisfy the constraints of the form while at the same inserts as many URLs pointing to our monitoring server as possible. As the implementation of the form-filling was contributed by Robin Kirchner, we omit further details about the algorithm here and instead refer the interested reader to our publication [176].

6.4 Large-Scale Study

Running a fully-featured browser on the server-side is a relatively recent phenomenon, likely caused by the rise in complexity in modern websites. The popularity of the now-defunct *PhantomJS* and its de-facto successor *Puppeteer* highlight the relevance of their use cases. However, the extent of usage and the security implications of running such an SSB as a service, i.e., where the attacker fully controls the URL destination, have not yet been studied. To gain insight into this phenomenon, we conduct a large-scale study on SSBs in the wild to gain insight into their prevalence and patch level. In the following, we describe our data collection and analysis steps in detail and then report on our results.

6.4.1 Data Collection and Post-Processing

For this study, we visited the 100,000 most popular websites according to the Tranco list [139] generated on March 2, 2021. On each website, we visit same-site links up to a depth of 10 or until we visited 50 pages, whichever comes first. If there are more than 50 same-site links on the landing page already, we randomly select 50 from among them. On each page, our crawler waits up to 30 seconds for the load event to trigger, otherwise, we flag the site as failed and move on. After the load event, we wait up to 3 more seconds for pending network requests to resolve to better handle pages that dynamically load additional content. We started 60 parallel crawlers using an instrumented Chromium 89.0.4389.72 on March 3 and finished the crawl about one week later on March 11. Of all the sites of the initial 100,000, we could only successfully visit about 79%. Of those that failed, about 8% were due to network errors, in particular, the DNS lookup often failed to resolve. In another 4%, the server returned an HTTP error code on the initial front page already. Additionally, 5.5% of these sites redirected to another domain which we subsequently discarded, since they are either duplicates like `blogger.com` and `blogspot.com`, or redirect to a location that is not part of the top 100k and thus out of scope. The remaining 3.5% failed due to various other issues, like failing to load before our 30 seconds timeout hit. In total, we successfully visited around 2.6M pages on about 79k sites. On these, we discovered 22.2M forms and submitted about 2.5M of them, the rest were considered duplicates. Additionally, we sent a total of 18M modified GET requests to about 5.6M different URLs in an attempt to discover SSRs that are triggered by URL parameters.

Regarding the post-processing of the data, we first of all only recorded requests to subdomains with a unique ID generated for each possible SSR candidate on each website that we visited. Thus, generic requests from scanners and crawlers are not part of this data.

For a meaningful analysis, we also have to prevent that our data is dominated by a few big companies, as they sometimes offer third-party scripts that trigger SSRs, e.g., DoubleClick and WordPress. These would then cause an incoming request any time we visit a domain that includes one of their affected scripts. Therefore, we use the *target domain* for attribution, e.g., if we submit a form included on *a.com* with an action URL to *b.com* that triggers an SSR, we consider *b.com* as the target domain and thus responsible for the incoming request, as it does matter less where we found this form and more where we submitted its data to. The same applies to SSRs triggered by modified URL parameters of third-party scripts.

Moreover, since these third parties are by nature present on a lot of websites, they would in total also send the most requests by far. As these companies also usually have access to vast IP ranges and heavily distribute their workload, we apply further deduplication based on the autonomous system number (ASN) — and not based on the IP address. Therefore, we define *unique requests* as those requests where the tuple *<target domain, asn, user agent>* is unique. If there are multiple, non-unique requests we use the fastest and ignore the rest. Additionally, we specifically exclude all requests from the *Googlebot* user agent, since their analytics products are widely used and result in many SSB visits from Google servers that would skew the analysis. Finally, we only consider incoming requests that we received within one week of visiting the page. Otherwise, popular websites with a high rank, i.e., sites that we visited at the very beginning, would have had almost twice as much time to send an SSR than websites which we visited towards the end of our one-week-long crawl. Obviously, we continued the data collection for one week after we had visited the last page with our crawler.

6.4.2 All Incoming Requests

In total, we recorded over 168,000 incoming requests, as Table 6.2 shows. Of these requests, we only consider 11,367 requests as unique according to our definition described in the previous subsection. The JavaScript usage is quite low when looking at all recorded requests with 4.5%, however this is to be expected since by far not all automated requests need the capabilities of a full browser. Yet these 7,500 requests with JavaScript enabled already cover around 17% of the unique requests, as the total number of requests is heavily skewed towards a few third-party services that still often use simple SSR implementations without a real browser. Overall, we triggered requests on 4,850 different domains of the initial 100,000. Therefore, our experiments enticed about 6% of the successfully crawled 79,000 sites to visit us back at a URL we presented. And about 16% of these domains even did so with a real browser that has JavaScript execution enabled.

Looking into the temporal dimension, we found that around 35% of all requests arrived within 1 minute after we had visited their page, as Fig. 6.3 shows. Yet, these 35% of requests in the first minute already cover about 50% of all domains due to repeated visits. Thus, on about 50% of sites where we could trigger *any* SSR, we had at least one visit *within the*

Table 6.2: SSRs recorded during our large-scale study.

	# Total	# with JavaScript
All requests	168055	7503 (4.5%)
Unique requests	11367	1973 (17.4%)
Unique domains	4850	760 (15.7%)
Unique IPs	8636	1571 (18.2%)
Unique AS	917	610 (66.5%)

first minute. On the other hand, the same is only true for 22% of the sites that visited us with JavaScript enabled. There are multiple reasons for this, e.g., SSBs could operate a bit slower than plain SSR implementations, and there are also likely humans still distorting the data, who manually visit our website much later after discovering our URL in their logs.

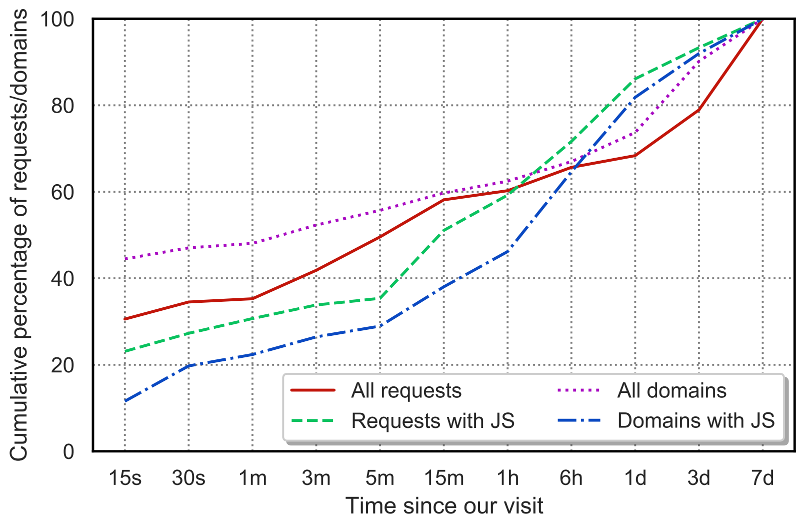

Figure 6.3: Incoming requests accumulated over time. Note the non-linear time axis.

6.4.3 Prevalence of SSBs

So far, we analyzed all incoming requests that we recorded. This also contains simple SSRs without a real browser, or from real browsers controlled by humans instead of automated systems. For all following analyses, we will now focus exclusively on SSBs, i.e., *real, automated browsers* with JavaScript enabled. As described in Section 6.2, discerning SSR tools without JS from real browsers with JS is straight-forward. On the other hand, discerning bots from humans with only a single request and without interaction is much more difficult. In order to do so nevertheless, we use the time difference between our visit to their website and their visit to our monitoring server and only consider cases where this time difference was less than 3 minutes, as such a fast reaction is a strong indicator for automated behavior. Moreover, as previously described, we also extend this time frame if additional bot indicators are present.

Table 6.3: Prevalence of server-side browsers on the left and their causes on the right-hand side.

	# Requests
All requests	3264
Unique requests	532
Unique IPs	440
Unique AS	206

	# Domains
All domains	254 (100.0%)
Header	58 (22.8%)
Param	34 (13.4%)
Form	167 (65.7%)

As already shown in Figure 6.3, focusing on only those requests that have JavaScript enabled and happened within in the first few minutes after our visit to their server reduces the total number of requests significantly. As Table 6.3 shows, 532 unique requests on 254 different domains were caused by bots running an SSB. Of these, 433 unique requests on 192 domains were labeled as bots because they arrived within 3 minutes. The remaining 99 requests on an additional 62 domains were labeled as bots due to a combination of a time threshold and our additional bot indicators. Moreover, Table 6.3 also shows the *type of triggers*, i.e., whether the visit was caused by the referer header we sent, a modified URL parameter, or a form submission. As the table shows, forms were most successful in attracting SSBs, being responsible for about 65% of all cases.

Next, we analyze the characteristics of the affected websites themselves and found that SSB implementations were significantly more common on the 10,000 most popular websites with 51 SSBs compared to the average of 23 SSBs per 10,000 sites in the remaining 90,000 sites. We found 30 of them within the top 5,000 and 14 of them even within the top 1,000 of the most popular websites according to the Tranco ranking. Moreover, we also investigated what types of categories these 254 sites belong to, using the *WebPulse Site Review* service [245] operated by the security company Symantec. As Table 6.4 shows, these sites are mostly related to technology and business.

Table 6.4: Categories of the sites with an SSB.

Category	# Sites	Category	# Sites
Analytics	7	News	14
Business	57	Other	39
Entertainment	3	Shopping	25
Education	21	Technology	69
Government	8	Uncategorized	11

Besides the websites themselves, we also investigated the origin of these SSB connections. We found that despite our deduplication, for some websites we received multiple, unique requests from real browsers even within the first few minutes. On 63 out of the 254 domains, we received requests from IP addresses belonging to two or more different ASNs. 20 of these even connected from 4 or more different ASNs and in one extreme case, one website even caused incoming connections from 22 different ASNs. On the other hand, we also found some networks that are responsible for a greater number of incoming connections for *different domains*. The three most popular ones were *AS8075* with connections triggered on 35 different domains, *AS15169* with 34 different domains, and *AS14618* and *AS16509* each with 18 domains. Closer investigation reveals that all these ASNs are related to cloud hosting, the first is owned by Microsoft and used for Azure, the second by Google for their cloud platform, while the other two are part of Amazon's AWS. Therefore, we can not assume a direct relation between these incoming requests, as many different companies likely just rely on the same, third-party hosting provider.

6.4.4 Investigation of Browser Versions

The 532 unique requests by bots (with JavaScript enabled and arriving within our time threshold) were conducted by 157 different HTTP user agents (UAs). The supposedly most popular used browser versions were *Chrome 84* in 122 requests, *Chrome 85* in 48 requests, and *Chrome 88* in 34 requests. However, only 64 requests had an HTTP UA that additionally clearly indicated that a bot is visiting us. For example, by either containing a reference to their service like *SpeedCurve* [230] in 3 cases or containing a general reference to an automation framework like *Headless Chrome* in 36 cases, *PhantomJS* in 7 cases, and *Lighthouse* [89] in 6 cases as the most popular tools in our data set.

As previously outlined, this UA information in the HTTP header is easily spoofed and should not be trusted blindly. Thus, we next compare these supposed values to the UA and platform information provided by JavaScript's `navigator` object. Looking into our collected data, we find that 13 unique requests definitely lied to us based on a mismatch between the HTTP UA and the JavaScript UA. Moreover, we also find that another striking 124 cases where the two UAs actually *do match* with each other but do not match with the platform information reported in the JavaScript environment. For example, these have a UA starting with *Mozilla/5.0 (Windows NT 10.0; Win64; x64)*, but `navigator.platform` reports *Linux x86_64* as their operating system. Table 6.5 list the most common examples of UA and platform mismatches.

These findings not only confirm that these are indeed bots, but also that they take some efforts to stay undetected by spoofing both UA string values. As expected, these bots with UAs claiming to be running on a Windows PC, an iPhone, or iPad, are actually all running on a Linux server, the preferred distribution for servers running automated tasks. With 137 requests with obviously spoofed information in total, about *one quarter* of the unique bot requests lied about their user agents, confirming that this voluntarily provided infor-

Table 6.5: Most frequently faked user agent strings (abbreviated) and the reported but mismatching platform

# Req.	HTTP Header	Platform
17	CPU iPhone OS 13_7 [...] Version/13.1.2	Linux x86_64
9	Windows NT 6.1 [...] Chrome/83.0.4103.106	Linux x86_64
9	Windows NT 6.1 [...] Firefox/77.0	Linux x86_64
7	Windows NT 10.0 [...] Chrome/79.0.3945.79	Linux x86_64
4	iPad; CPU OS 11_4 [...] Version/11.0	Linux x86_64

mation should indeed not be relied upon. Moreover, these findings should only be seen as a lower bound, as the value of `navigator.platform` could obviously be fake, as well.

To analyze their browsers in greater detail, we instead conduct a *feature fingerprinting* of all visitors, as described in Section 6.2. This way, we can determine their UA in an objective manner, without relying on potentially spoofed user agent strings. For 282 of the 532 unique requests, the version determined by the fingerprinting indeed did match the version of their HTTP and JavaScript UA value. In this case, we define *match* as within one major release as the fingerprint might not always be distinguishable for browsers with a fast release cycle. Of the remaining 250 requests where the user agents did not match our feature fingerprint, 80% claimed to be older than our fingerprint determined them to be with 201 requests, while 41 requests came from browsers claiming to be newer than our fingerprinting determined them to be. The remaining 8 requests did not send a browser version in their user agent information at all. Additionally, of those 124 requests that were previously found to be lying about their OS, in only 12 cases did their provided UA browser version match the results of our fingerprinting. This means, that in the remaining 112 cases, their browser version was apparently manipulated, too.

As described in Section 6.2, we then use the newest version derived from the three UAs (HTTP, JavaScript, fingerprint) as a conservative estimate for the actually used browser version. Table 6.6 shows the results, in which Chrome 84 is the most popular browser version with 150 unique requests. Combined with the other outdated, but popular versions Chrome 85 and Chrome 86, these three already make up for over 60% of all unique requests that we received. Chrome 88, which was the latest stable version of Chrome at the time of our crawl, only was responsible for 100 (19%) of SSB requests. In the next section, we will discuss the consequences to the security of all these servers running outdated SSBs in detail.

6.4.5 Vulnerable SSBs in the Wild

While there is a certain risk that even a fully up-to-date browser is exploited [206], in our scenario, we instead focus on outdated browsers in the wild. In the following, we describe our methodology to determine which browsers were vulnerable at the time of our crawl

Table 6.6: Number of requests by the five most popular user agents in SSBs. When the indicated versions differ, we use the newest of the three as the resulting user agent.

Browser	Indicator			Resulting UA
	HTTP Header	JavaScript	Fingerprint	
Chrome 88	34	29	112	100
Chrome 86	2	2	84	84
Chrome 85	48	48	48	95
Edge 85	12	12	0	12
Chrome 84	122	127	204	150

and how realistic it is to obtain a working exploit for them. Since 93% of the 532 requests were using Google Chrome or a Chromium-based browser like Edge, we will only discuss the security of different versions of this browser in detail.

Google released the latest stable Chrome version 89 for all platforms on March 2, i.e., one day before we started our crawl. This update contains 8 security fixes with a high severity, however they only started to roll out the update “over the coming days/weeks” [37]. The previous version 88 had been available for several weeks already and also contains many important security fixes, some of which earned a bug bounty of 10,000 dollars or more [36]. However, at time of writing, the details of these recently patched vulnerabilities in Chrome 88 and 89 had not yet been revealed in Google’s bug tracker [e.g., 43], therefore we do not know if the bug can be abused without user interaction. Nevertheless, reverse engineering patched software to create a working 1-day exploit is, in general, easier than searching for 0-day vulnerabilities from scratch [189]. As major browsers are nowadays open-source software, attackers do not even need to employ *binary diffing* techniques [e.g., 79, 26, 163] but can directly look a human-readable diffs including detailed comments about the changes. As previous research has shown, sometimes it might even be possible to automatically discover the relevant security patches among the huge list of changes [276], as well as automatically create exploits from the identified patches [29].

However, for vulnerabilities patched with the release of Chrome 87 and earlier, even the full details of the bugs including proof of concept (PoC) exploits and a discussion by Chrome engineers were already publicly available at the time of our crawl [e.g., 44]. Thus, while Chrome 87 and 88 could likely be exploited by a skilled attacker reverse-engineering the patches, we nevertheless use a very conservative estimate here and only consider browser versions to be vulnerable if publicly disclosed, detailed information about their vulnerabilities exists. Therefore, we consider version 86 and all older versions of Chrome to be vulnerable at the time of our crawl. This also applies to Microsoft Edge, which is based on Chromium and shares their version naming scheme, and Puppeteer, which uses Headless Chrome internally.

Table 6.7: The five most popular SSBs with the number of requests and affected domains in our study. For reference, we also include their release date and a high/critical CVE with a PoC exploit for that specific version.

Browser	Requests	Domains	Release	CVE	PoC
Chrome 88	100	83	01/21	—	—
Chrome 86	84	44	10/20	2020-16015	[44]
Chrome 85	95	39	08/20	2020-6575	[41]
Edge 85	12	10	08/20	2020-6575	[41]
Chrome 84	150	68	07/20	2020-6559	[42]

With this information about the security of these different releases in mind, we now come back to the 532 unique requests. Of these, 405 (76%) were conducted with a browser version vulnerable to publicly available exploits, resulting in 168 out of the 254 domains with SSBs to be vulnerable. This means, that *two out of three* SSB implementations that visit and subsequently execute arbitrary, attacker-controlled JavaScript code, are running a severely outdated and vulnerable browser with publicly disclosed PoC exploits. As shown in Table 6.7, the most popular browser versions were already quite dated during our experiments in March 2021 and many of them were released more than *half a year ago*. For reference, the table also includes three CVEs for the most popular browsers versions we encountered. All of these three CVEs work on all major platforms, including Linux, are exploitable without user interaction, and have their details including a PoC exploit publicly available. When looking at Figure 6.4, we see that even in the top 10,000 domains, over half of the websites with an SSB are vulnerable. All in all, this demonstrates that server-side browsers pose a considerable risk to any organization that makes use of them and is a, so far, widely overlooked problem.

6.5 Related Work

In this section, we first discuss related works from the area of browser fingerprinting and bot detection. Then, we introduce previous work on the topic of server-side requests and discuss how it relates to our work.

6.5.1 Browser Fingerprinting and Bot Detection

Web fingerprinting as a means to recognize repeated visitors, i.e tracking users without storing a client-side state such as cookies, has been studied for over two decades [e.g., 151, 64, 152, 2, 70]. However, in contrast to these previous works, detecting *reoccurring visits* by the same browser is not in our scope and we also can not rely on longer-term behavioral analysis [120, 46] of visitors in order to differentiate bots from humans. Furthermore, we

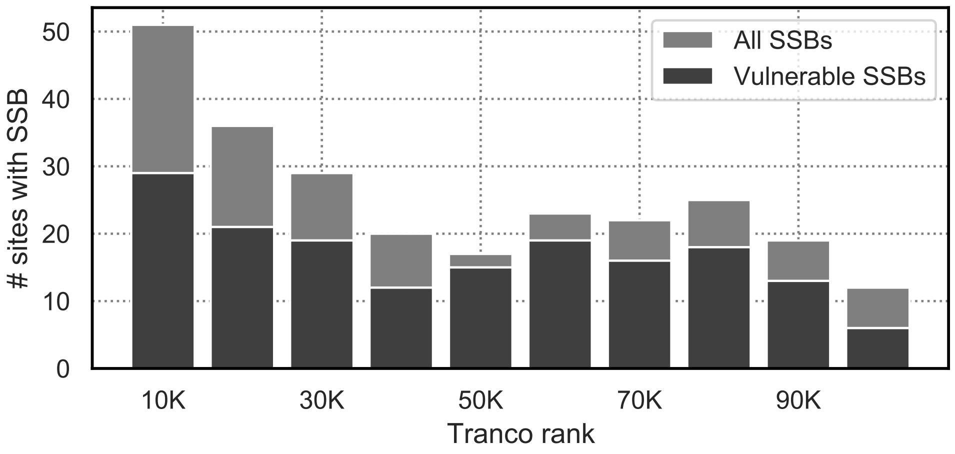

Figure 6.4: Distribution of vulnerable SSB implementations over Tranco ranks. The most popular sites are on the left.

can not resort to crawler-specific bot indicators, such as crawler traps [28, 258], or access log and traffic analysis [237, 273] on a series of requests, since the SSBs we target do not necessarily behave like crawlers. Instead, we are interested in identifying automated visitors with a real browser and their underlying browser version within a single request.

Thus, more closely related to our focus is research on detecting browser versions through JavaScript runtime and performance information [167, 107]. Unfortunately, these techniques suffer from long time requirements, making it impractical to analyze bots which, unlike humans, do not tend to leave browser tabs open for a long time. Moreover, approaches like Red Pills to detect virtualization are susceptible to processing and network bottlenecks which might introduce noise [107, 31]. Therefore, to infer the browser version, we instead leverage a JavaScript engine fingerprinting approach similar to Mulazzani et al. [173]. However, as previous work has shown, differentiating between bots and humans based on such fingerprints is problematic [258]. Yet in contrast to previous works we have additional knowledge about *when* to expect a visit and can identify bots based on this timing information.

6.5.2 Server-Side Requests

Generally speaking, server-side requests are one instance of a vulnerability class called *blind vulnerabilities.* One classic example of this class are *BlindSQL* [192] injections, where the attacker might not get a direct response of the query's results in text form, but can still infer their contents based on the application's response time to different queries. Another example of this are *Blind XML External Entities (XXE)* [229] attacks, in which links in XML sheets are abused to leak data. To aid the discovery of blind vulnerabilities, PortSwigger added the *Collaborator Everywhere* plugin [204] to their popular attack proxy *Burp Suite* [202].

Research specifically on the topic of server-side requests is rare. Stivala and Pellegrino

[240] studied how link previews on social media platforms can be manipulated to create benign-looking previews for malicious links. While the underlying link preview implementation makes use of server-side requests to fetch this information, the security of these implementations was not studied as part of their paper. In 2017, Orange Tsai [191] presented their findings on how differences in URL parsers cause filter bypasses leading to SSRF vulnerabilities in seemingly secure implementations. In 2021, Jabiyev et al. [119] performed a manual analysis of 61 HackerOne SSRF vulnerabilities and found that developer awareness for this vulnerability was still low. The authors propose a generic defense mechanism that proxies all SSRs through a helper server with no access to the internal network of that company, preventing the exfiltration of internal information. However, as all requests are forwarded through the proxy and still rendered and executed on the original, internal server, their defense would not protect against our attacker model.

Most closely related to our work is the 2016 publication titled "Uses and Abuses of Server-Side Requests" by Pellegrino et al. [198]. For their research, they developed *Günther*, a scanning tool that probes server-side backends for SSRF vulnerabilities. While they discuss the threat of the SSR implementation itself getting exploited, their attacker model in this case only considers DoS attacks, such as keeping the SSR provider busy with decompression tasks eventually leading to memory exhaustion. On the other hand, our work is, to the best of our knowledge, the first to study the consequences of running full *server-side browsers* with JavaScript execution in the wild. Unlike previous publications, we do not investigate traditional SSRF vulnerabilities and do also not try to circumvent filters or to confuse parsers. Instead, we use the SSR service *as intended* and instead directly attack those requesting clients that run a fully-featured, but outdated browser engine. Moreover, we are the first to systematically investigate this phenomenon on a large-scale and report on server-side requests conducted by the 100,000 most popular websites as compared to the previous studies on less than 100 websites.

7 Conclusion and Outlook

In this thesis, we looked at the unique challenges of detecting attacks and vulnerabilities on the *modern Web*. Specifically, we focused on automated security scanners, which enable analyses on a scale of millions of websites. We found that for multiple reasons, an instrumented browser is central to obtaining accurate results: First, websites increasingly rely on dynamically creating content on the client-side, which requires a browser and JavaScript engine to even correctly load these sites in the first place. Second, a heavy reliance on code provided by other parties blurs the line between first- and third-party code, which complicates reasoning about trust relationships and developer intent. Third, websites often behave non-deterministically due to volatile content, which prevents analyses that need to compare multiple executions of the same page in quick succession. Fourth, some of the attacks and vulnerabilities rely on emerging features that are not yet supported by emulated browsers or static analysis engines. Subsequently, we showed how to overcome these challenges and presented three concrete use-cases for our modern scanner based on an instrumented browser. Finally, we investigated the potential danger of using outdated browsers as part of an automated system. In the following, we summarize our main contributions but also discuss open questions and promising directions for future research related to this thesis.

Advancing the state of scanning Security scanners are an important tool for software engineers as well as researchers. In Chapter 2, we first created a traditional scanning pipeline based on Google's Tsunami, which can detect a fixed list of known vulnerabilities. Then, we discussed how to extend this pipeline to detect unknown instances of known *vulnerability classes* and the new crawling and analysis challenges this results in. Moreover, we introduced four web development trends, which further complicate the scanning of modern websites. Thereby, we concluded that emulating or re-implementing a browser is infeasible and that we instead need to integrate a real browser into the scanning pipeline. For this, we presented several browser *instrumentation approaches* and discussed their trade-offs. Using the example of SMURF, we then demonstrated how to solve one of the aforementioned challenges with our customized instrumentation code.

One of the open questions in working black-box security scanners is how to achieve a good scan coverage of the target. As we can only analyze the functionality that we discover during the crawling phase, a comprehensive crawl is paramount. In particular, very large websites with millions of subpages and websites that hide most functionality behind authentication or other hard-to-automate user interactions remain a challenge for current security scanners. While such *deep crawling* is a well-known problem in the area of content indexing [e.g., 190, 101, 103], security scanners have unique requirements and need to discover functionality over content. So far, few publications specifically focus on the needs of the security community in terms of deep crawling [71] and post-authentication crawling [122], necessitating further research on this topic.

Compatibility of an XSS defense Third-party integrations are a central building block of the Web. With ScriptProtect in Chapter 3, we presented a defensive mechanism that prevents attacks caused by benign-but-buggy third parties, without breaking functionality for most benign use-cases. For this, we used a transparent patching of dangerous JavaScript functions as a way to roll out our mechanism to all clients, without requiring changes to the browser. Then, we demonstrated how a modern web security scanner can be used in a *preventive* manner, i.e., by determining the compatibility of a defense on websites that are not yet vulnerable. As a result, we realized that sometimes a compromise between security and compatibility is necessary, and made ScriptProtect *slightly less strict* while simultaneously *vastly increasing its compatibility*. This way, we could prevent 90% of the attacks with a version of ScriptProtect that works out-of-the-box on about 30% of all websites.

While our mechanism to accurately track the initiators of dynamically included scripts works well in many cases, especially for large companies this does not completely solve the challenge caused by the *blurring of involved parties*. For example, an inclusion from two completely different domains like `google.com` and `youtube.com` might still actually represent a first-party inclusion, as both domains are owned and operated by the same entity. While SMURF [238] presents the notion of an *extended Same Party* that can deal with many of these issues, it nevertheless relies on various heuristics to group these domains. On the other hand, the working draft for *First-Party Sets* [262] proposes a standardized way for websites to mutually communicate which other domains should be considered to be first-party to them. Should this become a standard and find widespread adoption, it could supply a better ground truth for future research on the privacy and security of third-party code.

Abuses of JavaScript capabilities One of the best properties of the Web is that it is an *open* platform. Unlike other distribution mediums, the client-side part of a web application can be inspected by anyone without extra tools or a lot of special knowledge. However, as we showed in our work on anti-debugging techniques in Chapter 4, a significant number of websites would rather like to prevent that. We presented 9 anti-debugging techniques of varying severity, systematized them, and created an automated approach to detect all of them in the wild. Moreover and despite the general problem of *volatile content* on websites, we demonstrated how to use a web replay system to capture elusive techniques that make use of side-channels. Thereby, we again showcased the usefulness of a security scanner with an integrated browser that executes JavaScript, as a purely static detection of these techniques would have been much more difficult.

During this work, we operated under the assumption that attackers do not try to interfere with our attempts at detecting their anti-debugging techniques. So far, our detection approach thus relies on an implementation that could be detected by malicious scripts, e.g., by self-inspecting scripts that first probe their environment. A sensible next step would be to move our *in-band* JavaScript code to the *out-of-band* C++ realm, where it could not be directly observed by an attacker. Projects like VisibleV8 [123] seem to offer a promising way for researchers to achieve this without a deep understanding of the browser's code. Moreover, while we were successful in detecting these techniques, comprehensively *preventing* them is mostly an unsolved problem and likely requires modifying the browser and its underlying JavaScript engine. Therefore, we think a special *forensic browser* with countermeasures in place to enable safe and reliable debugging of client-side code in an adversarial setting would be a good avenue for further research.

Malicious WebAssembly With the rise of cryptocurrencies, monetizing malicious activity got significantly easier. Consequently, a considerable number of websites started to abuse their visitors by mining cryptocurrencies in their browser – without the need to infect them with malware. Given that these web-based miners rely on the support of multiple *emerging technologies* such as WebAssembly, we argue that accurately detecting and analyzing these attacks is only possible by using an instrumented browser that supports all these client-side technologies. In a broader sense, the introduction of WebAssembly also enabled new attacks and evasive techniques by going beyond the possibilities offered by the JavaScript language. During our investigation of the WebAssembly ecosystem in Chapter 5, we found the first indicators that it is already actively used to obfuscate code in an attempt to bypass adblockers and malware detectors.

One of the reasons for this shift in malicious activity was that, at least initially, a lot of tools and defensive mechanisms lacked support for this new language. For example, Romano et al. [216] recently showed how static JavaScript malware detectors that predated the introduction of WebAssembly had trouble detecting malicious code that made use of this new technology. While our early investigation of the WebAssembly ecosystem in 2018 was still based on manual analysis of the modules and the websites on which they were included, the security community slowly improved the tooling support over the years. Particularly noteworthy is *Wasabi* [141], a generic framework to dynamically analyze WebAssembly modules that laid the groundwork for further research. Moreover, the first fuzzers specifically modified to work well for WebAssembly [100, 142] were published at the end of 2021. Overall, the whole field of research and tooling support still appears to be in its infancy with regard to WebAssembly.

Security of automated browsers In this thesis, we established instrumented browsers as the central part of a modern web security scanner. They are also increasingly integrated into other automated systems due to the continually rising amount of client-side code on websites. However, browsers are one of the most complex pieces of software of our times, in which critical security bugs are found at a much higher rate than in other publicly exposed software. Therefore, in our final study in Chapter 6, we investigated how instrumented browsers are already used in the wild as part of other automated systems, where they present a unique attack surface by executing untrusted code on the server-side. Consequently, we revealed that *more than half* of the instrumented browsers that we discovered were running an outdated version and thus vulnerable to published proof-of-concept exploits.

In light of our findings, the decision of *not* automatically updating these server-side browsers by default should be reconsidered. On one hand, further work on mitigations, hardening, and isolation techniques [210, 211, 181, 129] could help prevent damage in case of an eventual browser compromise. Moreover, another approach would be to selectively disable unneeded features in an attempt to reduce the *exposed attack surface* [209, 208]. On the other hand, such mitigations only raise the bar for attackers without completely preventing the attack. Therefore, investigating how to best apply automatic browser updates to unattended systems without causing breakages would ultimately be the better solution.

To summarize, the contributions provided in this thesis advance the state of automated web security scanning and consequently help to make the Web a safer place.

Bibliography

[1] Gunes Acar, Steven Englehardt, and Arvind Narayanan. “No boundaries: data exfiltration by third parties embedded on web pages”. In: *Proc. of Privacy Enhancing Technologies Symposium (PETS)*. 2020.

[2] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. “FPDetective: Dusting the Web for Fingerprinters”. In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2013.

[3] AdGuard Research. *Cryptocurrency mining affects over 500 million people. And they have no idea it is happening.* Online `https://adguard.com/en/blog/crypto-mining-fever`. 2017.

[4] Adobe Corporate Communications. *Flash & The Future of Interactive Content.* Online `https://theblog.adobe.com/adobe-flash-update`. 2017.

[5] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. “JSand: complete client-side sandboxing of third-party JavaScript without browser modifications”. In: *Proc. of Annual Computer Security Applications Conference (ACSAC)*. 2012.

[6] Syed Taha Ali, Dylan Clarke, and Patrick McCorry. “Bitcoin: Perils of an Unregulated Global P2P Currency”. In: *Security Protocols XXIII*. 2015.

[7] Joey Allen, Zheng Yang, Matthew Landen, Raghav Bhat, Harsh Grover, Andrew Chang, Yang Ji, Roberto Perdisci, and Wenke Lee. “Mnemosyne: An Effective and Efficient Postmortem Watering Hole Attack Investigation System”. In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2020.

[8] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[9] Anthony Lieuallen. *Greasemonkey*. Online `https://addons.mozilla.org/en-US/firefox/addon/greasemonkey`. 2019.

[10] April King. *Analysis of the Alexa Top 1M sites*. Online `https://pokeinthe.io/2019/04/04/state-of-security-alexa-top-one-million-2019-04`. 2019.

[11] Babak Amin Azad, Oleksii Starov, Pierre Laperdrix, and Nick Nikiforakis. “Web runner 2049: Evaluating third-party anti-bot services”. In: *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 2020.

[12] Ruediger Bachmann and Achim D Brucker. “Developing secure software: A holistic approach to security testing”. In: *Datenschutz und Datensicherheit (DuD)* (2014).

[13] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. “Efficient Detection of Split Personalities in Malware.” In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2010.

[14] Albert-László Barabási, Vincent W. Freeh, Hawoong Jeong, and Jay B. Brockman. “Parasitic computing”. In: *Nature* (2001).

[15] Muhammad Ahmad Bashir, Sajjad Arshad, Engin Kirda, William Robertson, and Christo Wilson. “How tracking companies circumvented ad blockers using websockets”. In: *Proc. of Internet Measurement Conference (IMC)*. 2018.

[16] Kayce Basques. *What’s New In DevTools (Chrome 60)*. Online `https://developers.google.com/web/updates/2017/05/devtools-release-notes`. 2017.

[17] Daniel Bates, Adam Barth, and Collin Jackson. “Regular expressions considered harmful in client-side XSS filters”. In: *Proc. of the International World Wide Web Conference (WWW)*. 2010.

[18] BBC News. *British Airways fined £20m over data breach*. Online `https://www.bbc.com/news/technology-54568784`. 2020.

[19] Zahra Behfarshad and Ali Mesbah. “Hidden-web induced by client-side scripting: An empirical study”. In: *International Conference on Web Engineering*. 2013.

[20] Souphiane Bensalim, David Klein, Thomas Barber, and Martin Johns. “Talking About My Generation: Targeted DOM-based XSS Exploit Generation using Dynamic Data Flow Analysis”. In: *Proc. of European Workshop on System Security (EUROSEC)*. 2021.

[21] Eric Bidelman. *Getting Started with Headless Chrome*. Online `https://developers.google.com/web/updates/2017/04/headless-chrome`. 2017.

[22] Prithvi Bisht and VN Venkatakrishnan. “XSS-GUARD: precise dynamic prevention of cross-site scripting attacks”. In: *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 2008.

[23] Black Duck Open Hub. *Chromium Open Source Project*. Online `https://www.openhub.net/p/chrome/analyses/latest/languages_summary`. 2020.

[24] Kevin Borgolte, Christopher Kruegel, and Giovanni Vigna. “Meerkat: Detecting website defacements through image-based object recognition”. In: *Proc. of USENIX Security Symposium*. 2015.

[25] P Bourque and R Dupuis. *Guide to the Software Engineering Body of Knowledge Version 3.0*. Online `http://www.computer.org/web/swebok`. 2014.

[26] Martial Bourquin, Andy King, and Edward Robbins. “BinSlayer: accurate comparison of binary executables”. In: *Proc. of the ACM SIGPLAN Program Protection and Reverse Engineering Workshop (SSPREW)*. 2013.

[27] Mike Bowler. *HTMLUnit: A GUI-Less browser for Java programs*. Online `https://htmlunit.sourceforge.io`. 2021.

[28] Douglas Brewer, Kang Li, Laksmish Ramaswamy, and Calton Pu. “A Link Obfuscation Service to Detect Webbots”. In: *Proc. of IEEE International Conference on Services Computing*. 2010.

[29] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. “Automatic patch-based exploit generation is possible: Techniques and implications”. In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2008.

[30] Bugcrowd. *GitHub: HUNT - SSRF Python script*. Online `https://github.com/bugcrowd/HUNT/blob/master/ZAP/scripts/passive/SSRF.py`. 2018.

[31] Elie Bursztein, Artem Malyshev, Tadek Pietraszek, and Kurt Thomas. “Picasso: Lightweight Device Class Fingerprinting for Web Clients”. In: *Proc. of ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*. 2016.

[32] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. “Prophiler: a fast filter for the large-scale detection of malicious web pages”. In: *Proc. of the International World Wide Web Conference (WWW)*. 2011.

[33] Chrome Platform Status. *WebAssembly usage*. Online `https://www.chromestatus.com/metrics/feature/timeline/popularity/2237`. 2021.

[34] Chrome Platform Status. *WebSocket usage*. Online `https://www.chromestatus.com/metrics/feature/timeline/popularity/1149`. 2021.

[35] Chrome Platform Status. *WebWorker usage*. Online `https://www.chromestatus.com/metrics/feature/timeline/popularity/4`. 2021.

[36] Chrome Releases. *Chrome 88 Stable Channel Update for Desktop*. Online `https://chromereleases.googleblog.com/2021/02/stable-channel-update-for-desktop.html`. 2021.

[37] Chrome Releases. *Chrome 89 Stable Channel Update for Desktop*. Online `https://chromereleases.googleblog.com/2021/03/stable-channel-update-for-desktop.html`. 2021.

[38] ChromeDevTools. *Chrome DevTools Protocol*. Online `https://chromedevtools.github.io/devtools-protocol`. 2020.

[39] Chromium Blog. *Goodbye PNaCl, Hello WebAssembly!* Online `https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html`. 2017.

[40] Chromium Blog. *Speeding up Chrome’s release cycle*. Online `https://blog.chromium.org/2021/03/speeding-up-release-cycle.html`. 2021.

[41] Chromium Bug Tracker. *Issue 1081874: Double free on NodeChannel*. Online `https://crbug.com/1081874`. 2020.

[42] Chromium Bug Tracker. *Issue 1137630: PDFium heap-use-after-free*. Online `https://crbug.com/1137630`. 2020.

[43] Chromium Bug Tracker. *Issue 1138143: segmentation fault in mojom*. Online `https://crbug.com/1138143`. 2021.

[44] Chromium Bug Tracker. *Issue 1146670: TFC chrome full chain.* Online https://crbug.com/1146670. 2020.

[45] Chromium Bugtracker. *WebRequest API: allow extension to edit response body.* Online https://bugs.chromium.org/p/chromium/issues/detail?id=104058. 2011.

[46] Zi Chu, Steven Gianvecchio, Aaron Koehl, Haining Wang, and Sushil Jajodia. "Blog or block: Detecting blog bots through behavioral biometrics". In: *Computer Networks* (2013).

[47] Lin Clark. *What makes WebAssembly fast?* Online https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast. 2017.

[48] CoinMarketCap. *CoinMarketCap – Market Capitalization of Cryptocurrencies.* Online https://coinmarketcap.com/currencies. 2018.

[49] Bleeping Computer. *Massive Coinhive Cryptojacking Campaign Touches Over 200,000 MikroTik Routers.* Online https://www.bleepingcomputer.com/news/security/massive-coinhive-cryptojacking-campaign-touches-over-200-000-mikrotik-routers. 2018.

[50] Council of European Union. *Council regulation (EU) no 679/2016.* Online https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32016R0679. 2016.

[51] Marco Cova, Christopher Kruegel, and Giovanni Vigna. "Detection and analysis of drive-by-download attacks and malicious JavaScript code". In: *Proc. of the International World Wide Web Conference (WWW).* 2010.

[52] cure53. *DOMPurify Github Repository.* Online, https://github.com/cure53/DOMPurify. 2018.

[53] Charlie Curtsinger, Benjamin Livshits, Benjamin G Zorn, and Christian Seifert. "ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection." In: *Proc. of USENIX Security Symposium.* 2011.

[54] CVE Details. *CVE-2018-6140.* Online https://www.cvedetails.com/cve/CVE-2018-6140. 2019.

[55] CVE Details. *CVE-2019-11708.* Online https://www.cvedetails.com/cve/CVE-2019-11708. 2019.

[56] CVE Details. *CVE-2019-11752.* Online https://www.cvedetails.com/cve/CVE-2019-11752. 2019.

[57] CVE Details. *CVE-2019-5789.* Online https://www.cvedetails.com/cve/CVE-2019-5789. 2019.

[58] cyrus-and. *Chrome Remote Interface.* Online https://github.com/cyrus-and/chrome-remote-interface. 2021.

[59] donut. *HackerOne: SSRF on duckduckgo.com/iu/.* Online https://hackerone.com/reports/398641. 2018.

[60] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. “Enemy of the state: A state-aware black-box web vulnerability scanner”. In: *Proc. of USENIX Security Symposium.* 2012.

[61] Adam Doupé, Marco Cova, and Giovanni Vigna. “Why Johnny can’t pentest: An analysis of black-box web vulnerability scanners”. In: *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA).* 2010.

[62] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J Alex Halderman. “A search engine backed by Internet-wide scanning”. In: *Proc. of ACM Conference on Computer and Communications Security (CCS).* 2015.

[63] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. “ZMap: Fast Internet-wide scanning and its security applications”. In: *Proc. of USENIX Security Symposium.* 2013, pp. 605–620.

[64] Peter Eckersley. “How unique is your web browser?” In: *Proc. of Privacy Enhancing Technologies Symposium (PETS).* 2010.

[65] ECMA International. *ECMAScript 2009 Language Specification.* Edition 5. 2009.

[66] ECMA International. *ECMAScript 2015 Language Specification.* Edition 6. 2015.

[67] ECMA International. *ECMAScript 2017 Language Specification.* Edition 8. 2017.

[68] ECMA International. *ECMAScript 2019 Language Specification.* Edition 10. 2019.

[69] Electric Apps. *Vault AntiTheft.* Online `https://apps.shopify.com/vault-antitheft-protection-app`. 2020.

[70] Steven Englehardt and Arvind Narayanan. “Online Tracking: A 1-million-site Measurement and Analysis”. In: *Proc. of ACM Conference on Computer and Communications Security (CCS).* 2016.

[71] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. “Black Widow: Blackbox Data-driven Web Scanning”. In: *Proc. of IEEE Symposium on Security and Privacy (S&P).* 2021.

[72] Shayan Eskandari, Andreas Leoutsarakos, Troy Mursch, and Jeremy Clark. “A first look at browser-based cryptojacking”. In: *Proc. of IEEE Security and Privacy on the Blockchain Workshop.* 2018.

[73] Aurore Fass, Michael Backes, and Ben Stock. “JStap: a static pre-filter for malicious JavaScript detection”. In: *Proc. of Annual Computer Security Applications Conference (ACSAC).* 2019.

[74] Aurore Fass, Robert P Krawczyk, Michael Backes, and Ben Stock. “JaSt: Fully syntactic detection of malicious (obfuscated) JavaScript”. In: *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA).* 2018.

[75] Michael Felderer, Matthias Büchler, Martin Johns, Achim D Brucker, Ruth Breu, and Alexander Pretschner. “Security testing: A survey”. In: *Advances in Computers.* Vol. 101. 2016.

[76] Juan Manuel Fernández. *JavaScript AntiDebugging Tricks.* Online https://x-c3ll.github.io/posts/javascript-antidebugging. 2018.

[77] Ian Fette and Alexey Melnikov. *The WebSocket Protocol.* RFC 6455 (Proposed Standard). Online http://www.ietf.org/rfc/rfc6455.txt. Updated by RFC 7936. 2011.

[78] Firefox. *Remote Protocol.* Online https://firefox-source-docs.mozilla.org/remote/index.html. 2021.

[79] Halvar Flake. "Structural comparison of executable objects". In: *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA).* 2004.

[80] Mozilla Foundation. *Public Suffix List.* Online, https://publicsuffix.org. 2019.

[81] Github Pages. *SMURF Monitor Unveils Roadblocking Features.* Online https://smurf-ndss.github.io. 2021.

[82] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. "Undermining information hiding (and what to do about it)". In: *Proc. of USENIX Security Symposium.* 2016.

[83] Dan Goodin. *Now even YouTube serves ads with CPU-draining cryptocurrency miners.* Ars Technica, Online https://arstechnica.com/information-technology/2018/01/now-even-youtube-serves-ads-with-cpu-draining-cryptocurrency-miners. 2018.

[84] Google. *Chrome Release Cycle.* Online https://chromium.googlesource.com/chromium/src/+/master/docs/process/release_cycle.md. 2021.

[85] Google. *Puppeteer: Headless Chrome Node.js API.* Online https://github.com/puppeteer/puppeteer. 2021.

[86] Google. *Tsunami Security Scanner.* Online https://github.com/google/tsunami-security-scanner. 2021.

[87] Google. *Understanding your Chrome Browser update options.* Online https://services.google.com/fh/files/misc/chromeenterprisebrowser_updatestrategies_mktgwp_5.1.19.pdf. 2021.

[88] Google Developers. *Chrome DevTools.* Online https://developers.google.com/web/tools/chrome-devtools. 2019.

[89] Google Developers. *Lighthouse.* Online https://developers.google.com/web/tools/lighthouse. 2021.

[90] Google Git. *Web Page Replay.* Online https://chromium.googlesource.com/catapult/+/HEAD/web_page_replay_go. 2020.

[91] Google Groups. *Q4 2019 Summary from Chrome Security.* Online https://groups.google.com/a/chromium.org/g/security-dev/c/fbiuFbW07vI. 2021.

[92] Google Search Central. *Googlebot evergreen rendering in our testing tools.* Online https://developers.google.com/search/blog/2019/08/evergreen-googlebot-in-testing-tools. 2020.

[93] Robert David Graham. *MASSCAN: Mass IP port scanner.* Online https://github.com/robertdavidgraham/masscan. 2013.

[94] GreasyFork. *Anti Anti-debugger.* Online https://greasyfork.org/en/scripts/32015-anti-anti-debugger/code. 2017.

[95] Willem de Groot. *Cryptojacking found on 2496 online stores.* Online https://gwillem.gitlab.io/2017/11/07/cryptojacking-found-on-2496-stores. 2017.

[96] Web Incubator Community Group. *Explainer: Trusted Types for DOM Manipulation.* Online, https://github.com/WICG/trusted-types. 2017.

[97] Web Incubator Community Group. *Support application-specific sanitizers / type builders.* Online, https://github.com/WICG/trusted-types/issues/32. 2017.

[98] guya. *How to know when Chrome console is open.* Online https://blog.guya.net/2014/06/20/how-to-know-when-chrome-console-is-open. 2014.

[99] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. “Bringing the web up to speed with WebAssembly”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2017.

[100] Keno Haßler and Dominik Maier. “WAFL: Binary-Only WebAssembly Fuzzing with Fast Snapshots”. In: *Proc. of Reversing and Offensive-Oriented Trends Symposium (ROOTS).* 2021.

[101] Yeye He, Dong Xin, Venkatesh Ganti, Sriram Rajaraman, and Nirav Shah. “Crawling deep web entity pages”. In: *Proceedings of the sixth ACM international conference on Web search and data mining.* 2013.

[102] Mario Heiderich, Christopher Späth, and Jörg Schwenk. “DOMPurify: Client-Side Protection Against XSS and Markup Injection”. In: *Proc. of European Symposium on Research in Computer Security (ESORICS).* 2017.

[103] Inma Hernández, Carlos R Rivero, and David Ruiz. “Deep Web crawling: a survey”. In: *World Wide Web* (2019).

[104] Ian Hickson. *Web Workers.* W3C Working Draft. Online https://www.w3.org/TR/2015/WD-workers-20150924. 2015.

[105] Ariya Hidayat. *PhantomJS - Archiving the project: suspending the development.* Online https://github.com/ariya/phantomjs/issues/15344. 2018.

[106] Ariya Hidayat. *PhantomJS - Scriptable Headless WebKit.* Online https://github.com/ariya/phantomjs. 2016.

[107] Grant Ho, Dan Boneh, Lucas Ballard, and Niels Provos. “Tick Tock: Building Browser Red Pills from Timing Side Channels”. In: *Proc. of USENIX Workshop on Offensive Technologies (WOOT).* 2014.

[108] Grant Ho, Dan Boneh, Lucas Ballard, and Niels Provos. “Tick tock: building browser red pills from timing side channels”. In: *Proc. of USENIX Workshop on Offensive Technologies (WOOT)*. 2014.

[109] Geng Hong, Zhemin Yang, Sen Yang, Lei Zhang, Yuhong Nan, Zhibo Zhang, Min Yang, Yuan Zhang, Zhiyun Qian, and Haixin Duan. “How You Get Shot in the Back: A Systematical Study about Cryptojacking in the Real World”. In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2018.

[110] Andrew Horton and Brendan Coles. *WhatWeb*. Online `https://github.com/urbanadventurer/WhatWeb`. 2021.

[111] Danny Yuxing Huang, Hitesh Dharmdasani, Sarah Meiklejohn, Vacha Dave, Chris Grier, Damon McCoy, Stefan Savage, Nicholas Weaver, Alex C. Snoeren, and Kirill Levchenko. “Botcoin: Monetizing Stolen Cycles”. In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2014.

[112] Troy Hunt. *Have I Been Pwned*. Online `https://haveibeenpwned.com`. 2022.

[113] IANA. *IPv4 Address Space Registry*. Online `https://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xhtml`. 2021.

[114] Muhammad Ikram, Rahat Masood, Gareth Tyson, Mohamed Ali Kaafar, Noha Loizon, and Roya Ensafi. “The chain of implicit trust: An analysis of the web third-party resources loading”. In: *Proc. of the International World Wide Web Conference (WWW)*. 2019.

[115] Lon Ingram and Michael Walfish. “Treehouse: Javascript Sandboxes to Help Web Developers Help Themselves.” In: *Proc. of USENIX Annual Technical Conference (ATC)*. 2012.

[116] Luca Invernizzi, Paolo Milani Comparetti, Stefano Benvenuti, Christopher Kruegel, Marco Cova, and Giovanni Vigna. “Evilseed: A guided approach to finding malicious web pages”. In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2012.

[117] 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. of IEEE Symposium on Security and Privacy (S&P)*. 2016.

[118] Noriaki Iwasaki. *HackerOne: SSRF in Search.gov via ?url= parameter*. Online `https://hackerone.com/reports/514224`. 2019.

[119] Bahruz Jabiyev, Omid Mirzaei, Amin Kharraz, and Engin Kirda. “Preventing Server-Side Request Forgery Attacks”. In: *Proc. of ACM Symposium on Applied Computing (SAC)*. 2021.

[120] Gregoire Jacob, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. “PUB-CRAWL: Protecting Users and Businesses from CRAWLers”. In: *Proc. of USENIX Security Symposium*. 2012.

[121] Trevor Jim, Nikhil Swamy, and Michael Hicks. “Defeating script injection attacks with browser-enforced embedded policies”. In: *Proc. of the International World Wide Web Conference (WWW)*. 2007.

[122] Hugo Jonker, Stefan Karsch, Benjamin Krumnow, and Marc Sleegers. “Shepherd: a generic approach toautomating website login?” In: *Proc. of Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*. 2020.

[123] Jordan Jueckstock and Alexandros Kapravelos. “VisibleV8: In-browser Monitoring of JavaScript in the Wild”. In: *Proc. of Internet Measurement Conference (IMC)*. 2019.

[124] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. “Secubat: a web vulnerability scanner”. In: *Proc. of the International World Wide Web Conference (WWW)*. 2006.

[125] Alexandros Kapravelos, Marco Cova, Christopher Kruegel, and Giovanni Vigna. “Escape from monkey island: Evading high-interaction honeyclients”. In: *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 2011.

[126] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. “Hulk: Eliciting malicious behavior in browser extensions”. In: *Proc. of USENIX Security Symposium*. 2014.

[127] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. “Revolver: An Automated Approach to the Detection of Evasive Web-based Malware.” In: *Proc. of USENIX Security Symposium*. 2013.

[128] Manuel Karl, Marius Musch, Guoli Ma, Martin Johns, and Sebastian Lekies. “No Keys to the Kingdom Required: A Comprehensive Investigation of Missing Authentication Vulnerabilities in the Wild”. In: *Proc. of Internet Measurement Conference (IMC)*. 2022.

[129] Christoph Kerschbaumer, Tom Ritter, and Frederik Braun. “Hardening Firefox against Injection Attacks”. In: *Proc. of Workshop on Designing Security for the Web (SecWeb)*. 2020.

[130] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. “J-force: Forced execution on javascript”. In: *Proc. of the International World Wide Web Conference (WWW)*. 2017.

[131] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. “Barecloud: bare-metal analysis-based evasive malware detection”. In: *Proc. of USENIX Security Symposium*. 2014.

[132] Amit Klein. “DOM based cross site scripting or XSS of the third kind”. In: *Web Application Security Consortium, Articles* (2005).

[133] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. “Rozzle: De-cloaking internet malware”. In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2012.

[134] Radhesh K. Konoth, Emanuele Vineti, Veelasha Moonsamy, Martin Lindorfer, Christopher Kruegel, Herbet Bos, and Giovanni Vigna. “An In-depth Look into Drive-by Mining and Its Defense”. In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2018.

[135] Brian Krebs. *Who and What Is Coinhive?* Online `https://krebsonsecurity.com/2018/03/who-and-what-is-coinhive`. 2018.

[136] Amrit Kumar, Clément Fischer, Shruti Tople, and Prateek Saxena. “A Traceability Analysis of Monero’s Blockchain”. In: *Proc. of European Symposium on Research in Computer Security (ESORICS)*. 2017.

[137] Deepak Kumar, Zane Ma, Zakir Durumeric, Ariana Mirian, Joshua Mason, J Alex Halderman, and Michael Bailey. “Security challenges in an increasingly tangled web”. In: *Proc. of the International World Wide Web Conference (WWW)*. 2017.

[138] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. “Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web”. In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2017.

[139] 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. of Network and Distributed System Security Symposium (NDSS)*. 2019.

[140] Jiyeon Lee, Hayeon Kim, Junghwan Park, Insik Shin, and Sooel Son. “Pride and prejudice in progressive web apps: Abusing native app-like features in web applications”. In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2018.

[141] Daniel Lehmann and Michael Pradel. “Wasabi: A framework for dynamically analyzing webassembly”. In: *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019.

[142] Daniel Lehmann, Martin Toldam Torp, and Michael Pradel. “Fuzzm: Finding Memory Bugs through Binary-Only Instrumentation and Fuzzing of WebAssembly”. In: *arXiv preprint arXiv:2110.15433* (2021).

[143] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A Vela Nava, and Martin Johns. “Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets”. In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2017.

[144] Sebastian Lekies, Ben Stock, and Martin Johns. “25 million flows later: large-scale detection of DOM-based XSS”. In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2013.

[145] Nektarios Leontiadis, Tyler Moore, and Nicolas Christin. "Measuring and Analyzing Search-Redirection Attacks in the Illicit Online Prescription Drug Trade." In: *Proc. of USENIX Security Symposium*. 2011.

[146] Bo Li, Phani Vadrevu, Kyu Hyung Lee, Roberto Perdisci, Jienan Liu, Babak Rahbarinia, Kang Li, and Manos Antonakakis. "JSgraph: Enabling Reconstruction of Web Attacks via Efficient Tracking of Live In-Browser JavaScript Executions." In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2018.

[147] Timothy Libert. "An automated approach to auditing disclosure of third-party data collection in website privacy policies". In: *Proc. of the International World Wide Web Conference (WWW)*. 2018.

[148] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. "Detecting environment-sensitive malware". In: *Proc. of International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 2011.

[149] Giorgi Maisuradze, Michael Backes, and Christian Rossow. "Dachshund: digging for and securing against (non-) blinded constants in JIT code". In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2017.

[150] Srdjan Matic, Gareth Tyson, and Gianluca Stringhini. "Pythia: a Framework for the Automated Analysis of Web Hosting Environments". In: *Proc. of the International World Wide Web Conference (WWW)*. 2019.

[151] Jonathan R Mayer. "Any person... a pamphleteer": Internet Anonymity in the Age of Web 2.0". 2009.

[152] Jonathan R. Mayer and John C. Mitchell. "Third-Party Web Tracking: Policy and Technology". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2012.

[153] Judy McConnell. *WebAssembly support now shipping in all major browsers*. Online `https://blog.mozilla.org/blog/2017/11/13/webassembly-in-browsers`. 2017.

[154] MDN Web Docs. *performance.now()*. Online `https://developer.mozilla.org/en-US/docs/Web/API/Performance/now`. 2020.

[155] MDN Web Docs. *Proxy*. Online `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy`. 2019.

[156] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. "Riding out domsday: Towards detecting and preventing dom cross-site scripting". In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2018.

[157] Ali Mesbah, Engin Bozdag, and Arie Van Deursen. "Crawling Ajax by inferring user interface state changes". In: *2008 Eighth International Conference on Web Engineering*. 2008.

[158] Leo A Meyerovich and Benjamin Livshits. "ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2010.

[159] Microsoft Bing Blogs. *JavaScript, Dynamic Rendering, and Cloaking. Oh My!* Online `https://blogs.bing.com/webmaster/october-2018/bingbot-Series-JavaScript,-Dynamic-Rendering,-and-Cloaking-Oh-My`. 2021.

[160] Microsoft Windows Blogs. *A break from the past, part 2: Saying goodbye to ActiveX, VBScript, attachEvent.* Online `https://blogs.windows.com/msedgedev/2015/05/06/a-break-from-the-past-part-2-saying-goodbye-to-activex-vbscript-attachevent`. 2015.

[161] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. "Safe active content in sanitized JavaScript". In: *Google, Inc., Tech. Rep* (2008).

[162] MinerBlock. *minerBlock browser extension.* Online `https://chrome.google.com/webstore/detail/emikbbbebcdfohonlaifafnoanocnebl`. 2018.

[163] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. "BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking". In: *Proc. of USENIX Security Symposium.* 2017.

[164] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. "Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts". In: *Proc. of IEEE Symposium on Security and Privacy (S&P).* 2017.

[165] Malte Möser, Kyle Soska, Ethan Heilman, Kevin Lee, Henry Heffan, Shashvat Srivastava, Kyle Hogan, Jason Hennessey, Andrew Miller, Arvind Narayanan, et al. "An empirical analysis of traceability in the monero blockchain". In: *Proc. of Privacy Enhancing Technologies Symposium (PETS).* 2018.

[166] Alexander Moshchuk, Tanya Bragin, Damien Deville, Steven D. Gribble, and Henry M. Levy. "SpyProxy: Execution-based Detection of Malicious Web Content." In: *Proc. of USENIX Security Symposium.* 2007.

[167] Keaton Mowery, Dillon Bogenreif, Scott Yilek, and Hovav Shacham. "Fingerprinting Information in JavaScript Implementations". In: *Proc. of IEEE S&P Web 2.0 Security & Privacy Workshop (W2SP).* 2011.

[168] Mozilla. *Firefox Releases.* Online `https://www.mozilla.org/en-US/firefox/releases`. 2021.

[169] Mozilla. *GitHub: mdn/browser-compat-data.* Online `https://github.com/mdn/browser-compat-data`. 2021.

[170] Mozilla. *MDN Web Docs.* Online `https://developer.mozilla.org`. 2021.

[171] Mozilla. *Rhino ES2015/ES6, ES2016 and ES2017 support.* Online `https://mozilla.github.io/rhino/compat/engines.html`. 2021.

[172] Mozilla Developer Network. *Subresource Integrity.* Online `https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity`. 2019.

[173] Martin Mulazzani, Philipp Reschl, Markus Huber, Manuel Leithner, Sebastian Schrittwieser, and Edgar Weippl. "Fast and Reliable Browser Identification with JavaScript Engine Fingerprinting". In: *Proc. of IEEE S&P Web 2.0 Security & Privacy Workshop (W2SP)*. 2013.

[174] Paul Murley, Zane Ma, Joshua Mason, Michael Bailey, and Amin Kharraz. "WebSocket Adoption and the Landscape of the Real-Time Web". In: *Proc. of the International World Wide Web Conference (WWW)*. 2021.

[175] Marius Musch and Martin Johns. "U Can't Debug This: Detecting JavaScript Anti-Debugging Techniques in the Wild". In: *Proc. of USENIX Security Symposium*. 2021.

[176] Marius Musch, Robin Kirchner, Max Boll, and Martin Johns. "Server-Side Browsers: Exploring the Web's Hidden Attack Surface". In: *Proc. of ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 2022.

[177] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. "ScriptProtect: Mitigating Unsafe Third-Party Javascript Practices". In: *Proc. of ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 2019.

[178] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. "New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild". In: *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 2019.

[179] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. "Thieves in the Browser: Web-based Cryptojacking in the Wild". In: *Proc. of International Conference on Availability, Reliability and Security (ARES)*. 2019.

[180] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2009. URL: `http://www.bitcoin.org/bitcoin.pdf`.

[181] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. "Retrofitting fine grain isolation in the Firefox renderer". In: *Proc. of USENIX Security Symposium*. 2020.

[182] Christopher Neasbitt, Bo Li, Roberto Perdisci, Long Lu, Kapil Singh, and Kang Li. "Webcapsule: Towards a lightweight forensic engine for web browsers". In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2015.

[183] Netcraft. *January 2022 Web Server Survey*. Online `https://news.netcraft.com/archives/category/web-server-survey`. 2022.

[184] Dorothy Neufeld and Joyce Ma. *The 50 Most Visited Websites in the World*. Online `https://www.visualcapitalist.com/the-50-most-visited-websites-in-the-world`. 2021.

[185] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. "You are what you include: large-scale evaluation of remote javascript inclusions". In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2012.

[186] NoCoin. *NoCoin adblock list.* Online https://github.com/hoshsadiq/adblock-nocoin-list. 2018.

[187] NoCoin. *NoCoin browser extension.* Online https://chrome.google.com/webstore/detail/gojamcfopckidlocpkbelmpjcgmbgjcl. 2018.

[188] Jan Honza Odvarko. *Saying Goodbye to Firebug.* Online https://hacks.mozilla.org/2017/10/saying-goodbye-to-firebug. 2017.

[189] Jeongwook Oh. *Fight against 1-day exploits: Diffing Binaries vs Anti-diffing Binaries.* Online http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.694.8684&rep=rep1&type=pdf. 2009.

[190] Christopher Olston and Marc Najork. *Web crawling.* Now Publishers Inc, 2010.

[191] Orange Tsai. *A New Era of SSRF – Exploiting URL Parser in Trending Programming Languages.* Online https://www.blackhat.com/docs/us-17/thursday/us-17-Tsai-A-New-Era-Of-SSRF-Exploiting-URL-Parser-In-Trending-Programming-Languages.pdf. 2017.

[192] OWASP. *Blind SQL Injection.* Online https://owasp.org/www-community/attacks/Blind_SQL_Injection. 2021.

[193] OWASP. *OWASP Top 10.* Online https://owasp.org/Top10. 2021.

[194] OWASP. *WIVET: Web Input Vector Extractor Teaser.* Online https://github.com/bedirhan/wivet. 2014.

[195] Serkan Özkan. *CVE Details.* Online http://www.cvedetails.com. 2021.

[196] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. “Auto-patching DOM-based XSS at scale”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* 2015.

[197] Riccardo Pelizzi and R Sekar. “Protection, usability and improvements in reflected XSS filters.” In: *Proc. of ACM Asia Conference on Computer and Communications Security (ASIA CCS).* 2012.

[198] Giancarlo Pellegrino, Onur Catakoglu, Davide Balzarotti, and Christian Rossow. “Uses and Abuses of Server-Side Requests”. In: *Proc. of International Symposium on Research in Attacks, Intrusions and Defenses (RAID).* 2016.

[199] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. “jäk: Using dynamic analysis to crawl and test modern web applications”. In: *Proc. of International Symposium on Research in Attacks, Intrusions and Defenses (RAID).* 2015.

[200] Andrey Petukhov and Dmitry Kozlov. “Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing”. In: *Computing Systems Lab, Department of Computer Science, Moscow State University* (2008).

[201] Phu H Phung, David Sands, and Andrey Chudnov. “Lightweight self-protecting JavaScript”. In: *Proc. of ACM Symposium on Information, Computer and Communications Security (ASIACCS).* 2009.

[202] PortSwigger. *Burp Suite*. Online https://portswigger.net/burp. 2021.

[203] PortSwigger Ltd. *Cracking the lens: Targeting HTTP's Hidden Attack-Surface*. Online https://portswigger.net/research/cracking-the-lens-targeting-https-hidden-attack-surface. 2021.

[204] PortSwigger Ltd. *GitHub: collaborator-everywhere*. Online https://github.com/PortSwigger/collaborator-everywhere. 2021.

[205] Project Zero. *0day "In the Wild"*. Online https://googleprojectzero.blogspot.com/p/0day.html. 2021.

[206] Project Zero. *October 2020 0-day discovery*. Online https://googleprojectzero.blogspot.com/2021/03/in-wild-series-october-2020-0-day.html. 2021.

[207] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. "All Your iFRAMEs Point to Us". In: *Proc. of USENIX Security Symposium*. 2008.

[208] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. "Slimium: Debloating the Chromium Browser with Feature Subsetting". In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2020.

[209] Anh Quach, Aravind Prakash, and Lok Yan. "Debloating software through piece-wise compilation and loading". In: *Proc. of USENIX Security Symposium*. 2018.

[210] Charles Reis, Adam Barth, and Carlos Pizano. "Browser Security: Lessons from Google Chrome". In: *Queue* (2009).

[211] Charles Reis, Alexander Moshchuk, and Nasko Oskov. "Site isolation: Process separation for web sites within the browser". In: *Proc. of USENIX Security Symposium*. 2019.

[212] Andres Riancho. *W3AF: Web Application Attack and Audit Framework*. Online https://w3af.org. 2013.

[213] Konrad Rieck, Tammo Krueger, and Andreas Dewald. "Cujo: efficient detection and prevention of drive-by-download attacks". In: *Proc. of Annual Computer Security Applications Conference (ACSAC)*. 2010.

[214] Juan D Parra Rodriguez and Joachim Posegga. "RAPID: Resource and API-Based Detection Against In-Browser Miners". In: *Proc. of Annual Computer Security Applications Conference (ACSAC)*. 2018.

[215] Juan D. Parra Rodriguez and Jochaim Posegga. "CSP & Co. Can Save Us from a Rogue Cross-Origin Storage Browser Network! But for How Long?" In: *Proc. of ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2018.

[216] Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. "Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2022.

[217] Andreas Rossberg. *WebAssembly Core Specification*. W3C Working Draft. Online https://www.w3.org/TR/2018/WD-wasm-core-1-20180215. 2018.

[218] Jan Rüth, Torsten Zimmermann, Konrad Wolsing, and Oliver Hohlfeld. "Digging into Browser-based Crypto Mining". In: *Proc. of Internet Measurement Conference (IMC)*. 2018.

[219] Nicolas van Saberhagen. *CryptoNote v2.0*. Tech. rep. CryptoNote, 2013.

[220] Sansec. *Digital skimmer runs entirely on Google, defeats CSP*. Online `https://sansec.io/research/skimming-google-defeats-csp`. 2020.

[221] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. "FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications." In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2010.

[222] "Seigen", Max Jameson, Tuomo Nieminen, "Neocortex", and Antonio M. Juarez. *CryptoNight Hash Function*. CryptoNote Standard 008. 2008. URL: `https://cryptonote.org/cns/cns008.txt`.

[223] Selenium. *WebDriver documentation*. Online `https://www.selenium.dev/documentation/webdriver`. 2021.

[224] Tiago Serafim and Timofey Kachalov. *JavaScript Obfuscator Tool*. Online `https://obfuscator.io`. 2020.

[225] Shodan. *Search Engine*. Online `https://www.shodan.io`. 2021.

[226] Sindresorhus. *devtools-detect*. Online `https://github.com/sindresorhus/devtools-detect`. 2020.

[227] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. "Anything to hide? Studying minified and obfuscated code in the web". In: *Proc. of the International World Wide Web Conference (WWW)*. 2019.

[228] Peter Snyder, Cynthia Taylor, and Chris Kanich. "Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security". In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2017.

[229] Christopher Späth, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. "SoK: XML parser vulnerabilities". In: *Proc. of USENIX Workshop on Offensive Technologies (WOOT)*. 2016.

[230] SpeedCurve. *Monitor front-end performance*. Online `https://speedcurve.com`. 2021.

[231] StackOverflow. *Find out whether Chrome console is open*. Online `https://stackoverflow.com/questions/7798748`. 2011.

[232] StackOverflow. *How can I block F12 keyboard key*. Online `https://stackoverflow.com/questions/28575722`. 2015.

[233] StackOverflow. *How does Facebook disable the browser's integrated Developer Tools?* Online `https://stackoverflow.com/a/50674852`. 2014.

[234] StackOverflow. *How to detect Safari, Chrome, IE, Firefox and Opera browser?* Online `https://stackoverflow.com/a/9851769`. 2020.

[235] StackOverflow. *How to quickly and conveniently disable all console.log statements in my code?* Online `https://stackoverflow.com/questions/1215392`. 2009.

[236] Sid Stamm, Brandon Sterne, and Gervase Markham. "Reining in the web with content security policy". In: *Proc. of the International World Wide Web Conference (WWW)*. 2010.

[237] Athena Stassopoulou and Marios D. Dikaiakos. "Web robot detection: A probabilistic reasoning approach". In: 2009.

[238] Marius Steffens, Marius Musch, Martin Johns, and Ben Stock. "Who's Hosting the Block Party? Studying Third-Party Blockage of CSP and SRI". In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2021.

[239] 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. of Network and Distributed System Security Symposium (NDSS)*. 2019.

[240] Giada Stivala and Giancarlo Pellegrino. "Deceptive previews: A study of the link preview trustworthiness in social platforms". In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2020.

[241] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. "How the web tangled itself: Uncovering the history of client-side web (in) security". In: *Proc. of USENIX Security Symposium*. 2017.

[242] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. "Precise Client-side Protection against DOM-based Cross-Site Scripting". In: *Proc. of USENIX Security Symposium*. 2014.

[243] Ben Stock, Benjamin Livshits, and Benjamin Zorn. "Kizzle: a signature compiler for detecting exploit kits". In: *Proc. of Conference on Dependable Systems and Networks (DSN)*. 2016.

[244] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. "From facepalm to brain bender: Exploring client-side cross-site scripting". In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2015.

[245] Symantec. *WebPulse Site Review*. Online `https://sitereview.bluecoat.com`. 2020.

[246] Janos Szurdi, Balazs Kocso, Gabor Cseh, Jonathan Spring, Mark Felegyhazi, and Chris Kanich. "The long "taile" of typosquatting domain names". In: *Proc. of USENIX Security Symposium*. 2014.

[247] Rashid Tahir, Muhammad Huzaifa, Anupam Das, Mohammad Ahmad, Carl Gunter, Fareed Zaffar, Matthew Caesar, and Nikita Borisov. "Mining on someone else's dime: Mitigating covert mining operations in clouds and enterprises". In: *Proc. of International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 2017.

[248] Mike Ter Louw, Karthik Thotta Ganesh, and VN Venkatakrishnan. “AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements.” In: *Proc. of USENIX Security Symposium*. 2010.

[249] Mike Ter Louw and VN Venkatakrishnan. “Blueprint: Robust prevention of cross-site scripting attacks for existing browsers”. In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2009.

[250] The Monero Project. *Monero: Home*. Online `https://getmonero.org`. 2018.

[251] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon McCoy, Antonio Nappa, Vern Paxson, Paul Pearce, et al. “Ad injection at scale: Assessing deceptive advertisement modifications”. In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2015.

[252] Patrick Thomas. *BlindElephant*. Online `https://sourceforge.net/projects/blindelephant`. 2012.

[253] Time Magazine. *20 Most Influential Inventions of the 20th Century*. Online `http://content.time.com/time/photogallery/0,29307,2026224,00.html`. 2019.

[254] Tobias Urban, Martin Degeling, Thorsten Holz, and Norbert Pohlmann. “Beyond the front page: Measuring third party dynamics in the field”. In: *Proc. of the International World Wide Web Conference (WWW)*. 2020.

[255] Phani Vadrevu, Jienan Liu, Bo Li, Babak Rahbarinia, Kyu Hyung Lee, and Roberto Perdisci. “Enabling Reconstruction of Attacks on Users via Efficient Browsing Snapshots.” In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2017.

[256] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. “WebJail: least-privilege integration of third-party components in web mashups”. In: *Proc. of Annual Computer Security Applications Conference (ACSAC)*. 2011.

[257] Arie Van Deursen, Ali Mesbah, and Alex Nederlof. “Crawl-based analysis of web applications: Prospects and challenges”. In: *Science of computer programming* 97 (2015), pp. 173–180.

[258] Antoine Vastel, Walter Rudametkin, Romain Rouvoy, Xavier Blanc, and Antoine Vastel Datadome. “FP-Crawlers: Studying the Resilience of Browser Fingerprinting to Block Crawlers”. In: *Proc. of Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*. 2020.

[259] Amit Vasudevan and Ramesh Yerraballi. “Cobra: Fine-grained malware analysis using stealth localized-executions”. In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2006.

[260] VirusTotal. *Analyze suspicious files and URLs*. Online `https://www.virustotal.com/gui/home/url`. 2021.

[261] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. “Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis.” In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2007.

[262] W3C Privacy Community Group. *First-Party Sets*. Online `https://github.com/privacycg/first-party-sets`. 2021.

[263] W3C WebAssembly Community Group. *WebAssembly Design Documents*. Online `https://webassembly.org`. 2019.

[264] David Y Wang, Matthew Der, Mohammad Karami, Lawrence Saul, Damon McCoy, Stefan Savage, and Geoffrey M Voelker. “Search+ seizure: The Effectiveness of Interventions on SEO Campaigns”. In: *Proc. of Internet Measurement Conference (IMC)*. 2014.

[265] David Y Wang, Stefan Savage, and Geoffrey M Voelker. “Cloak and Dagger: Dynamics of Web Search Cloaking”. In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2011.

[266] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Sam King. “Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites that Exploit Browser Vulnerabilities”. In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2006.

[267] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W Hamlen, and Shuang Hao. “SEISMIC: SEcure in-lined script monitors for interrupting cryptojacks”. In: *Proc. of European Symposium on Research in Computer Security (ESORICS)*. 2018.

[268] Wasabi. *Dynamic Analyis Framework*. Online `http://wasabi.software-lab.org`. 2019.

[269] Wayback Machine. *Spotify*. Online `https://web.archive.org/web/20180301010204/https://www.spotify.com/us/`. 2018.

[270] 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. of ACM Conference on Computer and Communications Security (CCS)*. 2016.

[271] Michael Weissbacher, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. “Zigzag: Automatically hardening web applications against client-side validation vulnerabilities”. In: *Proc. of USENIX Security Symposium*. 2015.

[272] Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. “Comprehensive analysis and detection of flash-based malware”. In: *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 2016.

[273] Haitao Xu, Zhao Li, Chen Chu, Yuanmi Chen, Yifan Yang, Haifeng Lu, Haining Wang, and Angelos Stavrou. “Detecting and Characterizing Web Bot Traffic in a Large E-commerce Marketplace”. In: *Computer Security*. 2018.

[274] Wei Xu, Fangfang Zhang, and Sencun Zhu. “JStill: mostly static detection of obfuscated malicious JavaScript code”. In: *Proc. of ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2013.

[275] Wei Xu, Fangfang Zhang, and Sencun Zhu. “The power of obfuscation techniques in malicious JavaScript code: A measurement study”. In: *2012 7th International Conference on Malicious and Unwanted Software*. 2012.

[276] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. “SPAIN: Security Patch Analysis for Binaries towards Understanding the Pain and Pills”. In: *Proc. of International Conference on Software Engineering(ICSE)*. 2017.

[277] YouTube. *Tuned Cats*. Online `https://www.youtube.com/watch?v=lz7U_flXck4`. 2007.

[278] Jingyu Zhou and Yu Ding. “An analysis of urls generated from javascript code”. In: *2012 IEEE/ACIS 11th International Conference on Computer and Information Science*. 2012.